PROGRAMMING EASY WITH THE TI-99/4A

RICHARD GUENETTE
JAMES VOGEL

Easy Programming with the TI-99/4A

Easy Programming with the TI-99/4A

Richard Guenette
and
James Vogel

Birkhäuser
Boston • Basel • Stuttgart

Library of Congress Cataloging in Publication Data

Guenette, Richard.
Easy programming with the TI-99/4A

Bibliography: p.
Includes index.
1. TI 99/4A (Computer)—Programming. 2. Basic (Computer program language)
I. Vogel, James, 1952- . II. Title. III. Title: Easy programming with the
T.I.-99/4A.
QA76.8.TI33G83 1983 001.64'2 83-15714
ISBN 0-8176-3166-6

A B C D E F G H I J

ISBN 0-8176-3166-6

Printed in USA

Foreword

Welcome to the world of computer programming. Your Texas Instruments TI-99/4A is a real 16-bit microcomputer, complete with sound and color graphics. Making these features do what *you* want may seem impossible at first, but don't worry; there's nothing mysterious about computer programming. Learning to program computers simply means learning a new language—in this case, TI BASIC (for Beginner's All-purpose Symbolic Instruction Code), a version of the most popular language used on today's microcomputers. This book will teach you, step by step, how to tell your machine what you wish it to do—in other words, how to program it.

But that's not all. You'll also find chapters on general microcomputer principles, cassette deck use, the TI-99/4A as a terminal for much larger systems, options for expansion, and a list of resources for getting the most out of your home computer.

With the 99/4A you have access to a large library of programs, or software, already written by someone else. Without knowing anything about programming, you can use this software to play games, learn math, or store addresses. But at some point you'll want to make your personal computer really personal. You might want to create your own video game or compose a tune. And that's when you'll want to learn programming.

Learning to program is like learning to ride a bicycle: you must overcome the fear of falling. Once you do, you move right along. Once you overcome the fear of indecipherable computerese, you'll move right along in programming, too. And, as with bicycle riding, you learn by doing.

To get the most out of this book, you'll need your computer, a TV set (preferably color), a cassette tape recorder and connecting cable, your *User's Reference Guide,* and the confidence that *you* can do it.

Remember: computers, personal or otherwise, are machines. They are tools, extensions of the people who use them. Your TI-99/4A is an extension of you, and the programs you write are extensions of your own imagination. So let's begin.

Contents

Easy Programming with the TI-99/4A

1 Microcomputers in Brief

This chapter introduces microcomputers: their structure, methods of storing information, and languages. You'll find this information useful but not absolutely necessary for understanding how to program your TI-99/4A. If you're eager to grapple with the machine on your desk, skip right ahead.

OF BITS AND BYTES

On the most basic level, computers process information using numbers having only one digit. These numbers are called **binary numbers;** each digit is called a **bit** (short for "binary digit"). The value of a single bit can either be 1 or 0. To the computer, a 1 means an electrical "on"; a 0 means "off." All computer operations are ultimately manipulations of these binary numbers.

The basic building blocks, bits, are in turn assembled to form **bytes.** A byte is a series of eight bits. Each byte is equivalent to a letter or number, a keyboard symbol, or single character space in a page of text. The word "byte," for instance, would occupy four bytes. The byte representing the letter "A" is composed of the bits 01000001. Every time you press the letter "A" the computer translates it into this series of bits.

STORING INFORMATION

Computers store information in different kinds of memory. One kind is called **random access memory,** or RAM. RAM is like a blank sheet of paper that the computer allows you to write instructions, or programs, on. There are two types of RAM, **static** and **dynamic.** Static RAM, found

in some expensive computers, retains information if the power is shut off. Dynamic RAM, on the other hand, loses all its stored information if power is even briefly interrupted. Your 99/4A has enough dynamic RAM to hold 16,384 bytes of information. One thousand and twenty-four bytes is termed a **kilobyte,** or 1K; thus the 99/4A has 16K RAM.

Another type of memory is ROM, or **read only memory.** ROM, which cannot be changed or lost, determines many essential operating characteristics of a computer system. It typically stores the computer's **operating system** and BASIC language. ROM also stores the bytes that define each of the characters on the computer's keyboard. The 99/4A has 26K of ROM that contains TI BASIC, the graphics operating system, a system monitor to regulate the computer's overall performance, and the software for transferring information to and from a cassette deck.

For the record, two other kinds of computer memory are PROMs and EPROMs. PROMs are programmable read-only memories. EPROMs are PROMs which can be erased and re-programmed. PROM and EPROM memories function much like ROM.

ELEMENTARY COMPUTER ARCHITECTURE

The heart of a computer consists of the integrated circuit chips called microprocessors. Many microcomputers these days have more than one microprocessor. The 99/4A, for instance, has several: a main 16-bit microprocessor, plus one for operating the video display and another for generating sounds. The primary microprocessor, which coordinates the activity of the entire computer system, is called the CPU, or **central processing unit.**

Because a computer delivers processed information from data that has been put into it, it needs circuitry for performing input/output operations. The main console of the 99/4A has input/output circuitry to support several devices, or **peripherals:** the keyboard, joysticks, and a cassette recorder.

Thus, the computer is a collection of parts: a central processing unit and some lesser microprocessors dedicated to specific tasks, input/output circuitry to connect the computer to external devices, and memory for storing information. Other support circuitry and devices help these parts work together.

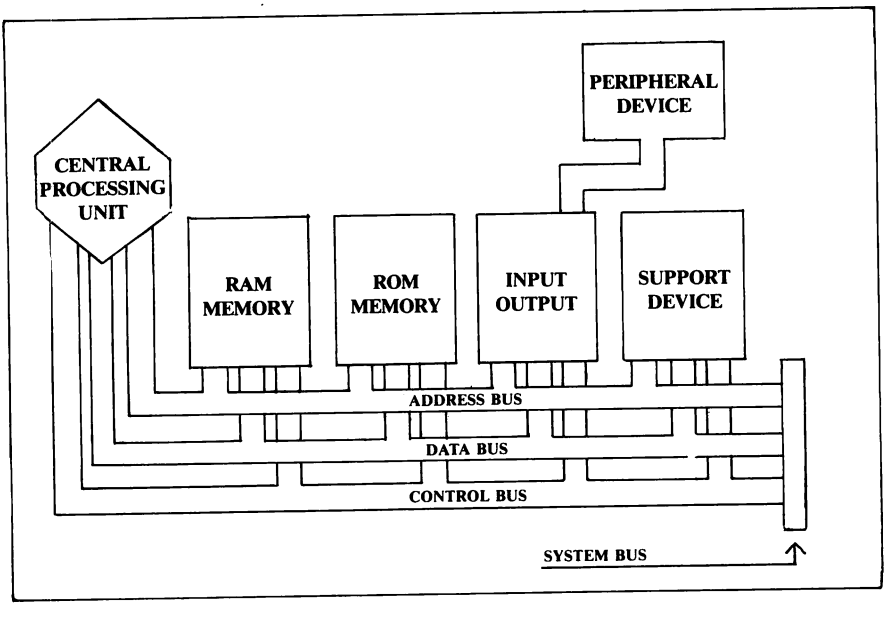
A system clock generates the timing needed to synchronize each section with the whole. A power supply powers the system and regulates the electrical voltages in the circuits. Interrupt circuitry interrupts the functioning of the CPU when an outside device, such as a joystick, is

operating. Finally, open pathways connect the computer's various parts so they can communicate. These pathways are called **buses.**

Buses come in several varieties. Two types deal directly with positioning information in memory: the **address bus** and the **data bus.** The address bus allows the CPU to determine the **address,** or memory location, in which to place or retrieve a piece of information. The data bus then sends the information to the appropriate address. In addition to the address and data buses the computer needs a **control bus,** which provides a pathway for control signals, such as those associated with input/output circuitry.

In many computers the address, data, and control buses are gathered into a single **system bus** that carries all the needed signals. A computer system can be extended by adding new circuitry to the system bus. The 99/4A's system bus allows you to extend your computer by plugging a peripheral expansion box into the slot provided on the right side of the main console (see Chapter 22, "System Options").

The diagram below shows a simplified version of general microcomputer architecture.



COMPUTER LANGUAGES

In the early days of computers, programs had to be written in a mode that was directly accessible to the computer. As computers only understand binary numbers, early programs appeared as long lists of ones and zeroes. Such programming proved tedious indeed. Since then, the science of software has given us a series of computer languages that have become increasingly easy for people to use and understand.

A hierarchy of computer languages has emerged which reflects the history of their development. We call the initial computer languages **machine language,** or **machine code,** and think of them as "low-level" languages. Although tedious to use, machine language is fast and powerful because it speaks directly to the CPU and requires no interpretation.

The next step toward a human-oriented language was **assembly language,** which allows programmers to write machine language programs by using symbolic names and abbreviations rather than strings of binary numbers. Although easier to use than machine language, assembly language is still not convenient for the average person to fool around with.

The computer languages most of us use today are "high-level" languages, such as BASIC, COBOL, PASCAL, FORTRAN, LOGO, LISP, or FORTH. These languages make programming easier because they use English-like commands that are relatively easy to understand. The disadvantage of these languages is that their programs must be translated into a form that can be used by the CPU which tends to make them slower and less powerful than machine language.

A high-level computer language can either be interpreted or compiled into machine language. Interpreted languages are translated into machine code line-by-line as the computer is running. Compiled languages are translated into machine code all at once before the program is run. Compiled programs run faster than interpreted ones but are harder to correct when something goes wrong.

The TI BASIC language covered in this book is an interpreted high-level language which represents years of work to make computer programming accessible to just about anybody. With a few keystrokes, you can achieve the same results that would have taken hours of labor in the first computer language.

2 Cassette Deck Notes

If you're just starting out with your 99/4A, you'll find that a cassette deck will give you a handy and inexpensive means of storing and retrieving information. This chapter contains equipment and maintenance suggestions and reviews the program loading and saving procedures for cassette decks.

CASSETTE DECKS

For easy operation and consistent information transfer, your tape deck should have:

- earphone (external speaker), microphone and remote jacks
- tone control
- a pause button
- a tape counter
- an AC power supply

The frequency response of the recording head and the steadiness of the tape transport mechanism may also affect performance. If you're having trouble, try cleaning and demagnetizing the tape head and cleaning the transport mechanism. Texas Instruments lists recommended tape decks in the literature that comes with your 99/4A.

For best results, adjust the tone control to maximum treble and the volume up half.

CASSETTE INTERFACE CABLES

A single cassette cable is all you need to store and retrieve programs.

In the past Texas Instruments marketed a dual cassette cable; they have replaced this product by a single cassette cable. Although a single cable is all you need, a dual cable allows you to use two cassette decks, making simultaneous input and output possible. The dual cables may not, however, be available on the market for long.

The ends of the dual cassette cable are marked **1** and **2**, for Cassette 1 (CS1) and Cassette 2 (CS2). Single cassette cables and the dual cable end marked 1 have three leaders, two that transfer information and one that enables the computer to turn the cassette deck on and off. CS1 is therefore used for loading and saving programs. The leads function as follows:

WHITE LEAD:	plugs into the EARPHONE jack; sends information *to* the computer.
BLACK LEAD:	plugs into the REMOTE jack; carries the switch that allows the computer to control the tape deck. The black lead is smaller than the other two.
RED LEAD:	plugs into the MICROPHONE jack; carries the signal *from* the computer.

Many different portable cassette decks will work with the 99/4A, but they must have these three jacks. The computer can send and receive information from a tape deck without a switching jack for the black lead, but you'll have to turn the tape deck on and off manually. Some tape decks have a switching jack with the wrong electrical polarity; then you must use an adapter or, again, work the deck manually.

LOADING AND SAVING PROGRAMS

The 99/4A gives you complete instructions right on the screen for loading and saving programs (see also page I-10 of your *User's Reference Guide*). You need only remember two commands:

SAVE CS1	(stores a program in memory onto tape).
OLD CS1	(reads a program on tape into memory).

To check for successful communication between computer and tape, simply type a "Y" (for "yes") when the computer asks you to CHECK TAPE (Y OR N)? The message DATA OK indicates that program transfer has been successful.

Be sure to type all cassette commands in upper-case letters (use the SHIFT key or depress ALPHA LOCK).

If a program fails to load, you will get one of two error messages:

ERROR IN DATA DETECTED (Volume too high, turn volume down)
ERROR—NO DATA FOUND (Volume too low, turn volume up)

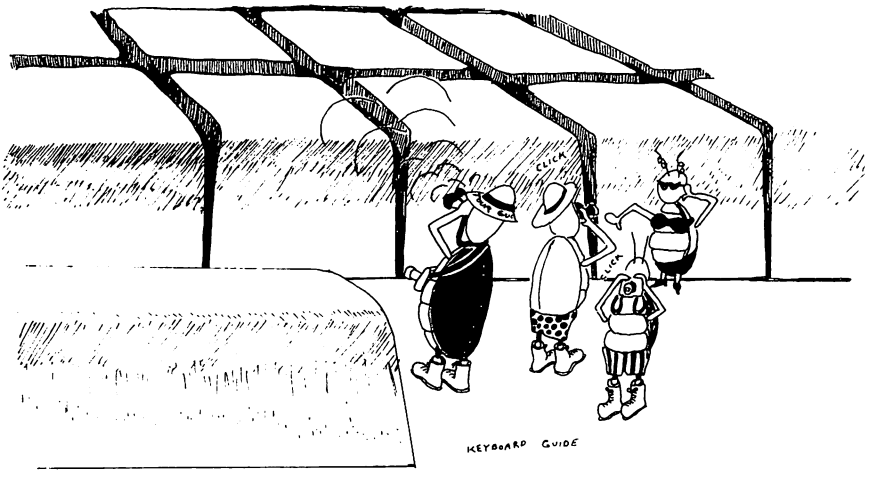
You'll also get the following list:

PRESS R TO READ (to try again)
PRESS C TO CHECK (to check the data)
PRESS E TO EXIT (to stop the loading procedure)

If you stop the loading procedure, the computer will ask you to press ENTER, issue a warning tone, and then display an I/O ERROR message. Check the I/O ERROR message list in the back of the *User's Reference Guide;* it may help solve your problem. If you try to reload your program, change the volume or tone settings on the tape deck.

Cassette tapes and decks can be quite finicky at times—with or without a computer. Keep your tapes away from any strong magnetic influence like a television. Do not use tapes longer than 30 minutes per side if you can help it—certainly no longer than 45 minutes a side. If you're still plagued by recording malfunctions, page I-2 of the *User's Reference Guide* has a good trouble-shooting list.

3 Keyboard Guide



The keyboard of the 99/4A produces 96 different symbols. These symbols include upper- and lower-case letters, numbers, numerical operators, punctuation marks, and the empty space character. The lower-case letters actually appear as small upper-case letters. Each key produces at least two symbols, and 12 of the keys produce three symbols. With a little practice, you will find the keyboard of your computer easy to operate.

ENTER KEY

ENTER is one of the most important and most often-used keys on the keyboard. Pressing it places what you type into the computer's memory. The computer cannot respond until you press ENTER; it promptly performs commands such as PRINT, LIST, RUN, or numerical operations when you do. When creating BASIC programs, you must

hit ENTER after every line. In short, you need the ENTER key for almost everything you do with your computer.

SHIFT AND ALPHA LOCK

Your keyboard has two SHIFT keys, located on either end of the second row from the bottom. These shift keys work like those of an ordinary typewriter to produce upper-case letters. The ALPHA LOCK key at the lower left corner corresponds to the shift-lock key on a typewriter, with one major difference. When ALPHA LOCK is up (off position), your keyboard will produce lower-case letters unless you SHIFT for upper-case; when it is down (or on), you will get only upper-case alphabetic letters regardless of the SHIFT key's position. Unlike the shift lock on a typewriter, ALPHA LOCK does not affect any keys other than the alphabetic keys. You will always get the symbols on the bottom half or numeric and punctuation keys unless you press SHIFT.

When writing a program, leave the ALPHA LOCK *down,* as the computer will not accept certain commands if they are written in lower-case. Moreover, when you LIST a program, your text will appear in upper-case letters unless it has been enclosed in quotation marks (more about LIST in Chapter 4). Type in the following lines *exactly* as shown; be sure to press ENTER after each line. If you make a mistake, use the FCTN and left arrow keys to backspace, then retype the correct text.

```
100 CALL CLEAR
110 FOR X=1 TO 5
120 PRINT "PRINT"
130 NEXT X
RUN                    (DON'T FORGET TO PRESS ENTER)
```

The computer prints the word "print" in lower-case letters five times. If you type LIST and press ENTER, the computer displays the program in upper-case except where the word "print" has been enclosed in quotes.

One quirk of the ALPHA LOCK key: when down, it can prevent upward movement on the screen in some programs. If program movements seem incorrect or non-existent, make sure the ALPHA LOCK is up.

SPECIAL SYMBOLS

In addition to the four arrow keys (E, S, D, and X), 12 other alphabetic keys have an extra character on their front faces. To produce these characters, hold down the FCTN (function) key in the lower right corner while pressing the symbol you want.

MATHEMATICAL OPERATORS

The keyboard has a number of keys for doing mathematical operations:

+	addition
–	subtraction
*	asterisk (multiplication)
/	slant (division)
=	equals
<	less than
>	more than
^	caret (to get exponents)

Of these characters, the caret is probably the least familiar. It tells the computer to raise a number to a power: to multiply the number by itself a certain number of times. Let's say you want to find out what three multiplied by itself four times is. You could ask the computer to:

```
PRINT 3*3*3*3
```

or

```
PRINT 3^4
```

In either case the answer is 81. By using the caret, however, you save yourself several keystrokes.

CONTROL KEYS

Your computer has several special control characters that are activated by simultaneously pressing the CTRL key to the left of the space bar. These control characters function primarily in formatting text when you are using your 99/4A as a terminal. To learn more about control characters, see Chapter 21, "Your Home Computer As A Terminal."

FUNCTION KEYS

A number of special keys enable you to move material around on the screen, edit programs, correct mistakes, and more. To use these function keys, first press the FCTN key and hold it down while pressing the desired function. We will indicate this operation with a colon—FCTN:1, for example.

If you haven't already done so, find the white plastic strip included with your 99/4A that carries the legend for function keys in the top row. Insert it into the tray above the keyboard. Now we can look at what each function key does.

FCTN:→. The right arrow on the D key lets you move the cursor to the right along a line without changing its text.

FCTN:←. Like the right arrow, the left arrow on the S key moves the cursor to the left along a line without changing it.

Use FCTN:→ together with FCTN:← to position yourself on a line when making insertions and deletions. Moving the cursor left and right is known as **scrolling** left or right.

FCTN:1 (DELete). Pressing FCTN:1 deletes characters from the screen. The delete function removes whatever character the cursor is positioned over when it is pressed. At the same time, all characters to the right of the deleted character are moved one space to the left. If you hold down the delete key, all the characters to the right of the cursor will disappear in rapid succession as if they were being sucked into the cursor. Type in some letters, use FCTN and left arrow to backspace to the beginning of your line, and try it.

FCTN:2 (INSert). To insert characters into the middle of a line, you must use the arrow keys to place the cursor one space to the right of the space where you wish to make your insertion. Then press FCTN:2

and type in your insertion. Anything to the right of your insertion will be bumped one space to the right to make room for each new character.

FCTN:↑. When you are working outside a program, i.e. in Immediate Mode, pressing the up arrow on the E key with the FCTN key has the same effect as pressing ENTER within a numbered program line, FCTN:↑ has special editing functions, which will be described in "Tools For Building Programs," Chapter 5.

FCTN:↓. This combination has exactly the same effect as FCTN:↑ has when used outside a program. We'll discuss these further in Chapter 5.

FCTN:3 (ERASE). Pressing the 3 key with FCTN erases the entire line you are on with a single keystroke.

FCTN:4 (CLEAR). This combination lets you "escape" from a line you are working on without having it registered by the computer: it performs the opposite action of the ENTER key. FCTN:CLEAR has additional uses described in Chapter 5.

FCTN:= (QUIT). Pressing FCTN:QUIT completely erases all information you've entered into the computer and returns you to the title screen. Anything displayed on the screen as well as all program contents are lost when you hit FCTN:QUIT so use it carefully!

Special Functions. The function keys BEGIN, PROC'D (proceed), AID, REDO, and BACK have special uses with various external software packages.

AUTOMATIC REPETITION

All the keys on the keyboard automatically repeat if held down. This feature makes editing or entering repetitious data much easier. For instance, you can hold down the space bar to quickly erase major sections of lines.

USING THE SCREEN EDITOR

Now let's have some fun working with the screen editor features that let you make sweeping changes quickly and easily. With ALPHA LOCK up, type in the following without hitting ENTER:

FOR SCORE AND EVEN EARS AGO

OUR OREFATHES HAD BOUGHT FOR

ONLY THIS HERE CONTNENT AN N

EW NATON. . .

Of course, this is a horribly mutilated version of the opening words of Lincoln's Gettysburg Address, which should read:

FOUR SCORE AND SEVEN YEARS

AGO OUR FATHERS BROUGHT

FORTH ON THIS CONTINENT A

NEW NATION. . .

First, let's make our deletions. The underlined letters below need to be deleted. Using the FCTN:← key and FCTN:→ keys, place the cursor over each space to be deleted and press the FCTN:DEL key as shown on your white function strip. Remember to delete the extra spaces between words where appropriate.

FOR SCORE AND EVEN EARS AGO

OUR OREFATHES HAD BOUGHT FOR

ONLY THIS HERE CONTNENT AN

NEW NATON. . .

You should now have:

FOR SCORE AND EVEN EARS AGO

OUR FATHES BOUGHT FORON THIS

CONTNENT A NEW NATON. . .

Now using FCTN:INS, insert the underlined letters in the version below. Again, use FCTN:← and FCTN:→ to position yourself within the line. Remember to place the cursor just to the right of the space into which you want to insert a character. Once in place, press FCTN:INS and type your insertion. Then reposition the cursor for your next insertion.

FOUR SCORE AND SEVEN YEARS AGO

OUR FATHERS BROUGHT FORTHON THIS

CONTINENT A NEW NATION. . .

We are most of the way there. We now have:

FOUR SCORE AND SEVEN YEARS

AGOOUR FATHERS BROUGHT FORTH

ON THISCONTINENT A NEW NATIO

N. . .

The last step in this process is to insert some spaces between words and at the ends of lines in order to give the passage a finished look. First insert a space between the words "ago" and "our." As this has placed the last letter in the word "forth" on the next line, we will place the entire word on the next line by inserting four spaces after "brought." Now break up "forthon" and "thiscontinent" with spaces and move the word "new" to the beginning of the fourth line. Lincoln should now be able to rest easy for you should see on your screen:

FOUR SCORE AND SEVEN YEARS

AGO OUR FATHERS BROUGHT

FORTH ON THIS CONTINENT A

NEW NATION. . .

By now you may be sick and tired of using the delete and insertions keys on this passage. If so, you may get your final revenge by practicing FCTN:ERASE and obliterating it.

4 Introducing TI BASIC

BASIC, which stands for "Beginners All-purpose Symbolic Instruction Code," was developed in the 1960s by John Kemeny and Thomas Kurtz of Dartmouth College. It is one of the most popular languages used on today's microcomputers. Your 99/4A comes with TI BASIC, a lightly modified version of the standard American National Standards Institute (ANSI) version.

Like any written language, TI BASIC has rules for spelling, grammar, syntax, and punctuation. These rules make it possible for the computer to understand what we want it to do. Unlike English, which has many thousands of words, TI BASIC has only seventy-nine, called **reserved words.**

If 79 new words sounds like a lot, don't worry; most of them do exactly what you would expect. For instance, NEW tells the computer that you are starting a new program: time to clear out the memory and get ready to begin. Similarly, END, when placed in a program, instructs the computer to stop the execution of that program.

Some BASIC words are specialized and rarely used; you will use others over and over again. Here and in following chapters, we will give you a working knowledge of most of the BASIC reserved words; you'll then be able to use your *Reference Guide* to figure out the remaining ones.

COMMANDS, FUNCTIONS, AND STATEMENTS

TI BASIC has three kinds of words: commands, functions, and statements. A **command** is an instruction that the computer performs as soon as you press ENTER. Commands exist outside of programs and allow you to communicate directly with the computer's operating system.

A **function** allows you to treat a complex operation—a procedure with many steps, such as the calculation of square roots or logarithms—as a single operation.

Statements are the fundamental building blocks of programs. A statement is an instruction to the computer that exists only inside a program; it is always preceded by a line number. A program, then, is simply a numbered list of statements written in BASIC.

Many BASIC words can operate either as commands or as statements. When they are used within a program, they act as statements; outside a program they act as commands for immediate response. The fold-out reference card accompanying your machine lists most TI BASIC words. Each is followed by a brief explanation and a symbol indicating whether it is a command, function, statement, or multi-category word.

COMMANDS

Immediate Mode

Let's try out a few commands. When you first press the 1 key to enter TI BASIC, your computer is in Immediate, or Command, Mode. In this mode, the computer *immediately* performs the BASIC commands that you type.

Be sure to type the following sample commands exactly as they appear here. Remember, BASIC has rules for spelling, punctuation, and spacing (see pages II-7 of the *User's Reference Guide*), and the computer is very fussy about details. When you are done, check your work and use the screen editor to change mistakes. When all is correct, press ENTER: the computer will then execute the command. If, for some reason, you have not correctly typed in the command, an **error message** like INCORRECT STATEMENT or BAD NAME may appear. If this happens, retype the command. We'll talk more about these error messages a little later (or see pages III-8 to III-12 of the *User's Reference Guide*).

Using The PRINT Command

You can print a message by typing the word PRINT, followed by a space and your message enclosed in quotation marks (FCTN:P) and ENTER:

```
PRINT "WORLDS ABOVE, WORLDS
BELOW"
```

You may PRINT more than one message at a time simply by enclosing each message in quotes. The messages may be separated by using one of three PRINT spacers. A colon places blank lines between each message. For example:

```
PRINT:::::"WORLDS ABOVE"::::
:"WORLDS BELOW"                (PRESS ENTER)
```

The computer will print WORLDS ABOVE five lines below the PRINT command and WORLDS BELOW five lines below WORLDS ABOVE.

Commas and semicolons are the other two print separators. A comma inserts a tab space between messages:

```
PRINT "WORLDS","APART"         (PRESS ENTER)
```

WORLDS appears at the left of the screen and APART appears on the same line but starting in the middle of the screen.

A semicolon instructs the computer to print the messages with no spaces between them, thereby joining the displays:

```
PRINT "WORLDS";"IN COLLISION"
```

will print WORLDSIN COLLISION.

Remember: PRINT separators must be placed outside the quotation marks; otherwise they will be considered part of the message's punctuation, like the comma in our first version.

CALL Commands

A quick look at the reference card shows a number of commands that begin with the word CALL followed by another word such as

SCREEN or KEY. Some of these words are also statements unique to TI BASIC. They are sometimes called the "knobs and whistles" of the 99/4A because of their dramatic effects. To see what we mean, type NEW to clear the screen and the following command (don't forget to press ENTER):

```
CALL HCHAR(5,7,42,10)
```

If you did this correctly, you should see a horizontal row of ten stars in the upper left of your screen. HCHAR stands for **horizontal character.** CALL HCHAR is a graphics command that instructs the computer to display the graphics encoded by the numbers in parentheses. CALL HCHAR can also double as a program statement. We'll show you how it is used and what those four numbers means in Chapter 15, "BASIC Graphics."

CALL HCHAR has a companion command, CALL VCHAR, which controls the vertical graphics display. To see it in action, type in the command below. Again, make sure the commas are in the right place and that all words are spelled and spaced correctly. Remember to press ENTER after you've checked your work.

```
CALL VCHAR(5,7,42,10)
```

Did you get a vertical column of ten stars at the upper left of your screen?

Another powerful command/statement is CALL SOUND. This command controls the output of the sound chip. To hear it in action, type:

```
CALL SOUND(2000,440,0)
```

You will hear a sound pitched at 440 cycles per second and lasting two seconds. The 99/4A has one of the easiest-to-use sound chips on the home computer market. It has three programmable voices. If you would like to hear them working together, type:

```
CALL SOUND(2000,440,0,294,0,
740,0)
```

We'll further explore the sound chip in Chapter 14, "Sound and Music." Sound and Music."

By now your screen is getting pretty crowded with commands. For a clean slate, type in this next command (remember to press ENTER):

```
CALL CLEAR
```

CALL CLEAR erases the screen display. It is often used as a program

statement, usually at the beginning, so that any new graphics or printouts will not be confused with previous displays.

Other CALL commands include CALL KEY, CALL SCREEN, CALL COLOR, CALL CHAR, and CALL JOYST. We will explain them all in later chapters.

Meanwhile, as long as we have a clean slate, let's go to the board and do a little math.

Command Mode Calculator

You can use the PRINT command along with the symbols for addition (+), subtraction (−), division (/), multiplication (*) and exponentiation (^) to turn your 99/4A into a calculator. The operations are simple. For addition, type in the following (don't forget to press ENTER):

```
PRINT 4+5
```

and you will get the answer 9 on the following line. The process is similar for the other mathematical operations:

```
PRINT 76-42

34

PRINT 25/5

5

PRINT 16*8

128

PRINT 9^2

81
```

Variables

In Immediate Mode, you can do calculations another way, by assigning a number value to a letter. For instance, if you type A = 3 and press the ENTER key, the letter A will be stored in memory as having a value of 3. You can now type the letter A in calculations, and the computer will automatically figure it in as 3. To see what we mean, type in the example below.

```
A=3

PRINT 4+5+A

 12
```

You have just assigned a value to a variable. A **variable** is a quantity—like the temperature – that varies. The letter A is the variable's name, and its value is 3. There are two types of variables in TI BASIC: **numeric,** defined in numbers, and **strings,** defined in characters. Variables play a critical role in programming because a simple name like A or B can represent a great deal of numeric information. Chapter 7, "Variables," will explain their use in more detail.

Your Immediate Mode calculator can do more than just simple operations. For example, try out the problem below. First try working it out with pencil and paper. Then use the 99/4A and see how fast the calculator works.

```
A=5

B=6

C=9

PRINT (A+B)+(A+C)-C/6

 23.5
```

Was your answer the same as the computer's? If not, check that you typed in all the parentheses; otherwise, the 99/4A may have performed the calculations in an order you did not expect. In all mathematical operations, the T.I. follows a very specific set of rules. Refer to Chapter 6, "Working with Numbers," or to page II-13 in your *User's Reference Guide.*

FUNCTIONS

You will recall that a BASIC function allows you to treat an operation involving a series of steps as a single procedure. What is actually happening is that your computer interprets your single word, such as SQR for square root, as a cue to perform a prescribed series of steps. Let's return to our calculator to see how this works.

Math Functions

Perhaps you remember Pythagoras's famous theorem on right triangles,

A∧2 + B∧2 = C∧2. Let's use this theorem and the BASIC function for finding square roots (SQR), to figure out the hypoteneuse of a right triangle with side A four inches long and side B five inches long.

The first step is to calculate the sum of A∧2 + B∧2.

```
A=4

B=5


PRINT A^2+B^2

41
```

We now know that our hypoteneuse (C), squared, equals 41. Do you know what the square root of 41 is? Your 99/4A does. Just type:

```
PRINT SQR(41)

6.4Ø3124237
```

Our hypoteneuse is 6.40312437 inches long.

Your computer has lots of built-in time-saving math functions that we'll discuss in Chapter 13, "Numeric Functions."

TAB Function

TI BASIC also has a function to help format PRINTed text. It's called TAB, and it works much like the tab key on your typewriter. To use it, simply type TAB and then, in parentheses, the number of spaces you wish to have your text moved. TAB function must be preceded and followed by a print spacer. The only exception to this rule is the first TAB, which needs no preceding spacer.

For a graphic demonstration, type in the line that appears below. You may want to clear your screen with a CALL CLEAR command first.

```
PRINT TAB(5);"UP"::TAB(1Ø);"

SIDE"::TAB(5);"DOWN"
```

The computer prints:

```
     UP

          SIDE

     DOWN
```

Notice that UP and DOWN, though on different lines, both begin on the fifth space while SIDE begins on the tenth. Print formatting is an important part of programming. You can learn more about it in Chapter 8, "Using PRINT."

STATEMENTS AND PROGRAM LINES

We come now to the last, but by no means the least important, BASIC word form: the statement. As mentioned before, statements are the building blocks of programs. Each statement tells the computer how to perform the work you want it to do. In TI BASIC only one statement may be written on a single numbered program line. The computer will execute each statement in numerical order, from lowest to highest.

To get a better picture of this, let's write a simple program that will clear the screen display and then PRINT a not-so-simple tongue-twister. Make sure when typing this program to leave a space between the line number and the BASIC statement (and keep the ALPHA LOCK key down). Press ENTER at the end of each line to store it in memory. When are you done ENTERing your lines, type RUN and hit ENTER to see your program at work.

One last tip: type NEW and hit ENTER before you begin. This will clear the memory of any leftover statements from old programs.

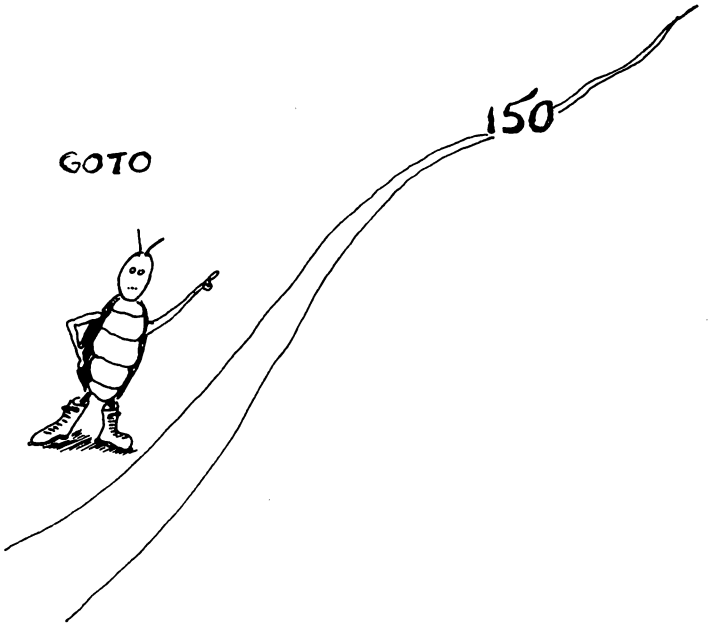
```
100 CALL CLEAR

110 PRINT "PETER PIPER"

120 PRINT "PICKED A PECK OF"

130 PRINT "PICKLED PEPPERS"
```

If all went well, the screen turned green, cleared the display, and printed your lines. The computer then printed DONE and the screen turned blue. DONE is the 99/4A's way of telling you it has finished executing your program. The green screen is the **default** screen color value when a program is running. In other words, unless you specify an alternative color in your program, the screen will always be green when it RUNs. The screen returns to blue when the final line has been executed.

The main difference between Immediate Mode and Program Mode is that in Program Mode, we can tell the computer to respond to more than just a single command.

If we wish to expand this program, we need only add more program lines. To demonstrate this, let's add some lines so that the computer PRINTs what happens when our tongue gets twisted. To do this we'll have to use a new statement. It's called GOTO, and it's one of the easiest-to-use BASIC program statements.

GOTO



As stated earlier, the computer will execute a program line-by-line in numerical order. One way to get around or change this order is to use the GOTO statement. You must always follow a GOTO with a line number, such as GOTO 12Ø. Make sure to leave a space between the GOTO and the number. When the computer encounters a GOTO, it will skip to the line indicated by the line number that follows the statement. Thus, GOTO 12Ø would instruct the computer to go to Line 12Ø. It would then execute Line 12Ø and move on to Line 13Ø again. Since GOTO can change the order of the lines, you can see why it would help demonstrate our tongue-twister.

Now let's change our program, which is still in memory. To make sure it's still there, type LIST and press ENTER. Add the new statements by typing in the lines that appear below. Don't worry if they seem out of order. The computer doesn't care whether you type the lines in the order they run; it has a built-in feature that automatically reorders program lines. Once again, make sure to press ENTER after you have typed each line.

```
105 GOTO 130
135 GOTO 120
125 GOTO 110
115 END
```

To check your work, type LIST and press ENTER. Your program should read:

```
100 CALL CLEAR
105 GOTO 130
110 PRINT "PETER PIPER"
115 END
120 PRINT "PICKED A PECK OF"
125 GOTO 110
130 PRINT "PICKLED PEPPERS"
135 GOTO 120
```

If you notice any mistakes, retype the line and press ENTER.

By numbering our original lines by tens, we made it easy to insert our new lines in between the old ones. We could, if we wanted, add Lines 101, 102, 103, etc. It's good programming practice to leave room for line additions; you never know when you will need the space.

Now, to see your tongue-twisted program, type RUN and press ENTER.

```
PICKLED PEPPERS

PICKED A PECK OF

PETER PIPER

** DONE **
```

GOTO has many other uses which we will explore a little later. Now that you've written your first program, you're ready for the next chapter, "Tools For Building Programs," but first, type in this final command, press ENTER and see what happens.

```
BYE
```

5 Tools for Building Programs

As we saw in the last chapter, commands that can stand alone are also tools for building programs. This chapter will review such commands and their roles in creating and editing programs. (Before beginning, be sure you're comfortable with the screen editor, described in Chapter 3 and in your *User's Reference Guide.*)

LINE NUMBERING

All programs are simply lists of numbered lines (instructions) that the computer performs one at a time. A program's logic depends, in part, on the sequence of those lines.

Your 99/4A has a simple labor-saving feature that lets you automatically establish line numbers for your program. To activate this feature, type NUMBER, or simply NUM. You are now working in **Number Mode.** The number 100 appears as your first program line entry because the **default value** of the NUMBER command produces a list of lines that begins at 100 and proceeds by increments of 10. (Default values are values that the computer assumes unless you tell it otherwise.)

If you wish, you may choose your own line numbers. Simply type NUM followed by the line number with which you wish to begin your program, then a comma, and the number of steps between line numbers. In the following example we will set up a line list that starts at 1000 and proceeds by steps of 20. Type in a letter and hit ENTER after each number as it appears on the screen:

```
NUM 1000,20
1000 A
1020 B
1040 C
1060 D
```

If you ENTER a blank line, you will leave the Number Mode and the computer will automatically stop providing new numbers.

If you wish to start at Line 100 but would like to proceed by fewer or more than 10 steps per line, type NUM followed immediately by a comma and the number of steps desired.

```
NUM,8
```

gives 100, 108, 116, 124, etc. Conversely, to start at any number but still proceed by steps of 10, type NUM followed only by a space and the starting line number. For example,

```
NUM 600
```

gives you 600, 610, 620, 630, etc.

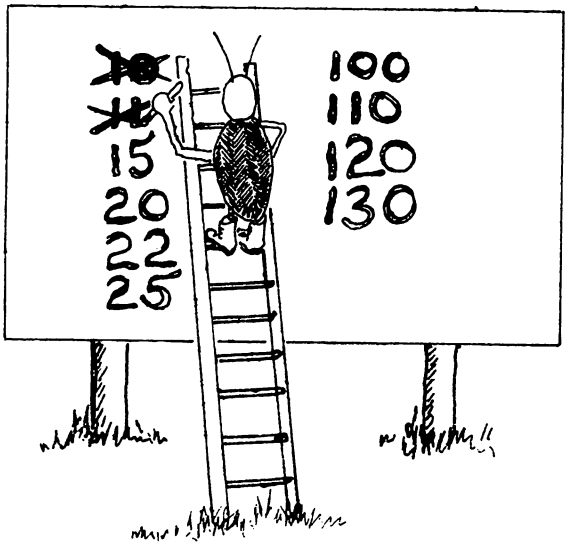
While in Number Mode, you may use the ENTER key to review an existing program. Say you've previously stored this program:

```
100 A=2
110 B=3
120 C=4
130 PRINT A+B+C
```

If you want to look at Line 110, you can call it by typing NUM 110. Each time you press ENTER the next program line will appear. You can then change any existing line or leave it unaltered. If you wish to exit the Number Mode at any time press FCTN:CLEAR If you edit a line, the new version replaces the old, and the computer displays the next line number in the sequence. If you want to erase a line from the program, you must use Edit Mode (see below).

In Number Mode, the INSert and DELete functions, as well as the left and right arrows, work exactly the same way as they do in the screen editor. The FCTN:↑ and FCTN:↓, however, work like the ENTER key: *both* arrows merely display subsequent numbered lines, just as the ENTER key does.

RESEQUENCE



A powerful programming tool in TI BASIC is the RESEQUENCE command. RESEQUENCE allows you to completely renumber all program lines with the touch of a few keys; this is a fabulous convenience when creating large, complex programs. To use the command, simply type RESEQUENCE or RES, followed by the number you wish to start with, a comma, and the number of steps desired between program lines. For instance, the command RES 5ØØ,5Ø changes the program on the left to the one on the right.

```
100 A=2
110 B=3
120 GOTO 190
130 C=4
140 D=5
150 GOTO 210
160 E=6
170 F=7
180 GOTO 230
190 PRINT A+B
200 GOTO 130
210 PRINT C+D
220 GOTO 160
230 PRINT E+F
```

```
500 A=2
550 B=3
600 GOTO 950
650 C=4
700 D=5
750 GOTO 1050
800 E=6
850 F=7
900 GOTO 1150
950 PRINT A+B
1000 GOTO 650
1050 PRINT C+D
1100 GOTO 800
1150 PRINT E+F
```

See for yourself: clear the computer by entering NEW, then use Number Mode to type in the program on the left. RUN it. Then type RES 500,50, hit ENTER and LIST your renumbered program.

When you RUN the new program you'll see that it does exactly the same thing as the original: it PRINTs the numbers 5, 9, and 13.

The computer takes only a moment to resequence such short programs, and perhaps as long as several minutes for very long ones. But think of the typing you save with this tool!

The RESEQUENCE command produces a program beginning at Line 100 and proceeding by 10s if used without a trailing number. Keep in mind that the maximum allowable line number is 32767; any request to RESEQUENCE a program that would give a line number higher than 32767 produces the error message, BAD LINE NUMBER.

You'll see the beauty of this command as you put together elements of complex programs. You might wish, for instance, to insert more pro-

gram steps in one spot than your current numbering system has room for. RESequencing offers a painless solution. It can also help clean up a program with a lot of odd line numbers.

EDITING PROGRAMS

Your 99/4A gives you several methods, besides the NUMBER command for reviewing and editing program lines. These methods, which use the LIST and EDIT commands as well the up and down arrows, will make creating programs much easier.

LIST

When you type LIST and press ENTER, the computer displays the program currently in memory, starting with the first line and proceeding until the last. If the program is longer than 24 lines, the first ones to appear will disappear off the top of the screen as the display continues. To stop a moving line list, simply press FCTN:CLEAR. The forms the LIST command can take are shown below:

LIST	Lists the entire program.
LIST 200–	Lists program lines from 200 onward.
LIST –200	Lists program lines up to and including Line 200.
LIST 200–500	Lists Lines 200 to 250, inclusive.
LIST 200	Lists only Line 200.

If you ask to LIST a line number higher than any actual number in a program, you'll get the last real line. Similarly, if you ask for a number lower than any in the program, you'll get the lowest numbered line. Asking for a line number greater than 32767 will bring a BAD LINE NUMBER message. Attempting to list a non-existent program will produce a CAN'T DO THAT message.

Edit Mode

To edit a program you will want to use Edit Mode. You may begin editing at any line number by typing EDIT followed by a space and the number of the line you wish to review. You can get the same effect with fewer keystrokes by typing in just the line number and then hitting either the up- or down-arrow functions. Thus, typing

```
EDIT 220
220 [FCTN: ↑ ]
220 [FCTN: ↓ ]
```

will all display Line 220 for you to review. Once the computer is in Edit Mode, the up and down arrows can be used to review a program line by line.

Type in the following program (clear out your old lines first with a NEW command):

```
100 PRINT "USE EDIT"
110 PRINT "MODE TO"
120 PRINT "CORRECT"
130 PRINT "YOUR PROGRAMS"
140 PRINT "EASILY"
150 [ENTER A BLANK LINE HERE.]
```

Typing 100 followed immediately by FCTN:↓ will put you in Edit Mode at Line 100. Pressing FCTN:↓ again will display the following line, number 110. If you hit FCTN:↓ three more times, the computer displays the next three lines in the program, ending with Line 140. Thus, FCTN:↓ moves you *down* through a program, displaying successively *higher* numbers, as if the lines were printed on a sheet of paper. Hitting FCTN:↑, on the other hand, displays the next *lower* line numbers (130, 120, 110, 100), effectively moving you *upward* through your program-on-paper. Note, however, that the cursor itself remains at the bottom of your screen while your lines move upward. If you run out of line numbers as you are ascending or descending the program list, you will automatically leave Edit Mode.

Moving through the line list with the up and down arrows allows you either to leave lines unchanged or to edit them. Any changes you make will replace the old version when you next press the arrows to continue. (The cursor may be anywhere on a line when you move to the next line.)

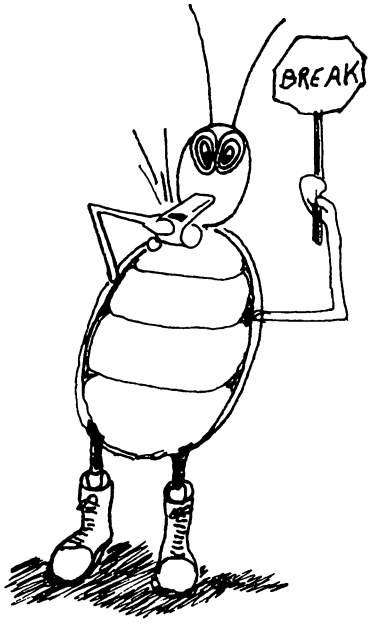
Thus, in Edit Mode FCTN:↓ and FCTN:↑ operate much like ENTER, entering revised statements into memory. Pressing ENTER itself also

enters changes in a program line but takes you out of Edit Mode. Use the arrow functions if you're planning further revisions.

You can INSert and DELete text with FCTN:← and FCTN: → in Edit Mode as described in Chapter 3, except that you won't be able to move the cursor over the line numbers themselves. In Edit Mode, you cannot immediately delete the line number even using FCTN:ERASE. Deleting all the text of a line will, however, delete the line number from memory. If you reLIST the program you'll see that the line number of the erased line has disappeared.

A very handy tool to use in Edit Mode is FCTN:CLEAR. This combination lets you escape from making changes in a line if you suddenly develop second thoughts. This is particularly useful when you're working on long and complicated lines or if you've covered some portion of the text with SSSSSSSS or DDDDDDDD by forgetting to press FCTN while trying to move left or right. No matter how badly you've mangled a line, you can always press FCTN:CLEAR and the computer will preserve the orginal line. FCTN:CLEAR does put you out of Edit Mode, but this is a small price to pay, as you need only type the line number followed by FCTN:↑ or FCTN:↓ to begin again.

STARTING AND STOPPING PROGRAMS



In this section we'll look at some commands for starting and stopping programs or parts of programs. This may sound trivial, but the ability to run small program sections independently will help you analyze and correct problems later.

Diamond Track

The following program graphically demonstrates the RUN, BREAK, UNBREAK, CONTINUE, STOP, and END commands. Typing it into the computer will also give you some practice with Edit Mode. "Diamond Track" is simply a group of print statements that have been run in a loop to produce a continuous diamond shaped track. While the program RUNs each line number is displayed on the right hand side of the screen as the line is executed. That way, you can keep track of the program as it RUNs. When typing, pay close attention to the blank spaces in each print statement; otherwise, who knows what shape you'll get?

```
2 REM *** DIAMOND TRACK ***
3 CALL CLEAR
6 PRINT TAB(20);"LINE #"::
10 PRINT "          A
   10"
20 PRINT "         B B
   20"
30 PRINT "        C   C
   30"
40 PRINT "       D     D
   40"
50 PRINT "      E       E
   50"
60 PRINT "     F         F
   60"
70 PRINT "    G           G
   70"
80 PRINT "     H         H
   80"
90 PRINT "      I        I
   90"
100 PRINT "       J     J
   100"
110 PRINT "        K   K
   110"
120 PRINT "         L L
   120"
130 PRINT "          M
   130"
140 GOTO 10
```

Okay, RUN your program. On the left, the Diamond Track will scroll up the screen with each diamond connected to the next where the M sits over the A. All the computer is doing is running a series of thirteen

PRINT program statements numbered 1Ø to 13Ø, over and over again.

Now, hit FCTN:CLEAR to stop the program and let's look at some commands.

RUN

You already know that typing RUN and pressing ENTER starts running the program stored in memory. When you type RUN alone, the computer starts the program at the lowest line number. If you add a line number to the command, the program beings at that line. If you now type RUN 7Ø, Diamond Track will begin at the widest point of the diamond by printing two Gs separated by eleven empty spaces. Notice that the number 7Ø appears on the right.

BREAK out of the program with FCTN:CLEAR; note the message BREAKPOINT AT [line number].

CONtinue

Like RUN, CONtinue also runs a program, but always from a BREAKPOINT where the program has been interrupted. To CONtinue, you need only type CON and press ENTER. The first line performed will have the same number as the number in the BREAKPOINT AT...message. Break out of Diamond Track several times and restart the program with the CON command.

The CON command will not work if you have edited any lines since the last breakpoint. To start the program after editing a line, use RUN.

BREAK and UNBREAK

Pressing FCTN:CLEAR when a program is RUNning stops it, but you can also use BREAK directly. For example, you may insert BREAK as a numbered statement in the program itself. Type BREAK into the Diamond Track program at a new line number—say, 55—and RUN it: the program stops after Line 5Ø with the message BREAKPOINT AT 55. Now bring 55 up in Edit Mode and change it to read: 55 BREAK 2Ø, 7Ø, 11Ø. Don't forget to put a space after the line number and after the word BREAK as well as separating with commas the line numbers including in the statement.

RUNning the program now produces breakpoints at Lines 2Ø, 7Ø, and 11Ø.

Use CON to continue after each breakpoint. You'll notice that the program doesn't stop at Line 2Ø the first time through because the com-

puter hasn't yet come to the BREAK statement at Line 55. The second time through the loop however, the program BREAKs at Line 2Ø. The values of variables in a program are not affected by a breakpoint.

Now, in Edit Mode, delete Line 55 with FCTN:ERASE. The program should again run without stopping.

Let's try using BREAK as a command. Hit FCTN:CLEAR to stop the program. Without a line number, type BREAK 4Ø, 6Ø and press ENTER. If you RUN the program now, you'll get a breakpoint first at Line 4Ø (type CON to go on) and then at Line 6Ø. Typing CON again will start an uninterrupted run: outside a program, the BREAK command breaks the program only once for each specified line number. To get repeated BREAKs, you must include BREAK as a numbered program statement.

If you ENTER a BREAK command for a non-existent line number, the message BAD LINE NUMBER is displayed immediately. You can enter a BREAK statement which refers to a non-existent line number, but the computer will simply beep a BAD LINE NUMBER warning when the BREAK statement is encountered and proceed with the program. If this happens, just delete or edit the BREAK statement.

UNBREAK removes breakpoints set by BREAK. UNBREAK may be used either by itself or with a list of line numbers it is intended to affect. For instance, UNBREAK 2Ø,4Ø would be used to delete breakpoints at Lines 2Ø and 4Ø. UNBREAK can be used as either a command or as a statement. To ensure the effectiveness of UNBREAK, it is best to use BREAK statements which refer to themselves such as

```
55 BREAK 55
```

Any number of BREAK statements written in this fashion can be cancelled by a simply UNBREAK command or statement written at the beginning of the program. BREAK statements which do not specify one or more line numbers cannot be cleared by UNBREAK. If you wish to clear only one BREAK statement, follow your UNBREAK statement or command with the line number in question. Following a BREAK command with an UNBREAK command which specifies no line numbers will cancel all breakpoints. If line numbers are specified with UNBREAK, only the breakpoints at the specified line numbers will be eliminated. If UNBREAK is used as a command, any breakpoint eliminated will be reinstated if they are encountered in the program more than once. To avoid this, use UNBREAK as a statement within the program.

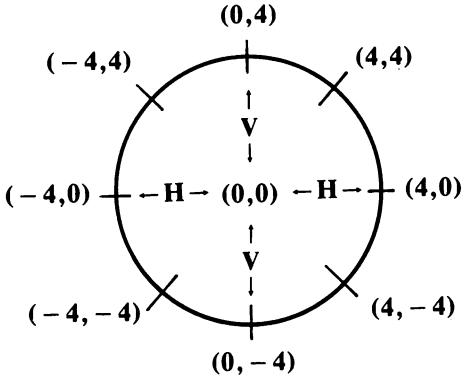
END and STOP

END and STOP statements do just that: a program stops running at the line saying END or STOP, and you get the message ** DONE **. You cannot CONtinue a program that has been ENDed or STOPped; use RUN to start again. In TI BASIC, END and STOP statements are not required to complete your programs.

REMARK STATEMENTS

You can write notes to yourself within a program by typing REM at the beginning of a number line. The computer completely ignores all REM lines when running a program. Note, therefore, that references within REMarks to other line numbers do not change when the RESEQUENCE command is used. Be sure to leave a space after REM before typing in your remarks.

6 Working with Numbers



TI BASIC has very precise ways of dealing with numbers. When assigning values to numerical variables, you can represent and enter them in two ways: as "real" numbers (integers and decimals) or in exponential (sometimes called scientific) notation.

Numbers like 22, .0648, −99, 46.583, and −3.14 are all "real" numbers. When entering large numbers, do not use commas or TI BASIC will interpret one number as two. Four thousand six hundred twenty-two must therefore be written 4622; the computer would read 4,622 as 4 and 622.

Negative numbers should be indicated by a minus sign before the number, for example, −25. If you don't add any sign, the computer will assume the number is positive, although you may put a plus (+) sign before a positive number if you wish.

Exponential, or scientific notation is usually reserved for very large numbers. 10^6, for example, means ten multiplied by itself six times (or raised to the power of six); in other words, one million. 2×10^6 means two times ten to the sixth power, or two million. Because you can't type superscripts on your 99/4A, 2×10^6 is written 2E6, where E stands for the 10. The number 3.4859E7 means, then, 3.4859×10^7 or 34,859,000.

Note that the E must be in upper case. The numbers before the E are known as the mantissa and usually range from 1.000 to 9.999. The number(s) following the E stand for the power of 10 to which you wish to raise your mantissa. Very small numbers have negative exponents; thus 4.982 X 10−4 is written 4.982E−4 and equals .000492.

If you enter a number that is too large for the computer to handle, or if a calculation results in such a number, you will see a "NUMBER TOO BIG" error message. Oh well, if you need a number larger than 9.9999999999999E127 or less than −9.9999999999999E127, you'd better get another computer.

ORDER OF ARITHMETIC OPERATIONS

TI BASIC performs arithmetic operations according to strict rules. The TI BASIC arithmetic operators are:

Operation	BASIC Symbol	Example
Addition	+	X+Y
Subtraction	−	X−Y
Multiplication	*	X*Y
Division	/	X/Y
Exponentiation	∧	X∧Y

When evaluating arithmetic expressions with more than one operation, the computer performs the operations in a specific order:

1. All expressions within parentheses are evaluated first; if parentheses are nested, the innermost expression will be evaluated first.
2. Exponents are evaluated next, from left to right.
3. Multiplication and division are next.
4. Addition and subtraction are last.

Thus the expression (4*(8/2)+6)*5 is evaluated as follows:

1. (8/2) = 4 comes first because it is the innermost expression in parentheses.
2. 4*(8/2) = 16 is performed next because, of the expressions remaining in the parentheses, multiplication takes priority over addition.
3. 4*(8/2) or 16 is then added to 6, as this is the next operation within parentheses, yielding a total of 22.
4. 22, as the total of all the operations performed within the parentheses is then multiplied by 5. The final solution is 110.

RELATIONAL EXPRESSIONS

The TI also has a set of relational expressions that indicate relations. These relational exprssions are usually teamed with the conditional logic of the IF-THEN-ELSE statement (see Chapter 9). The symbols below indicate the relationship between two numbers.

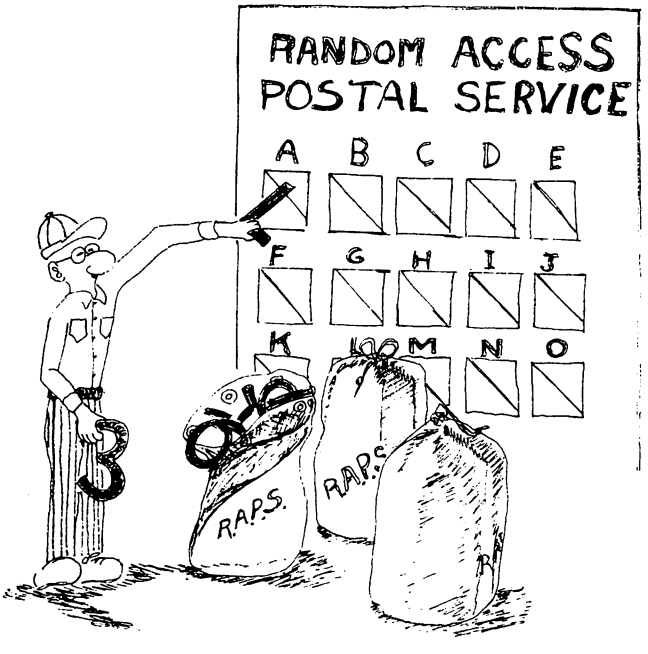
=	equal to
<	less than
>	greater than
< >	not equal to
< =	less than or equal to
> =	greater than or equal to

Relational expressions can also be used within numeric expressions. When a relational expression is used within a numeric expression, the numeric value of -1 is given if the statement is true, while the numeric value of 0 is given if the relation is false. To clarify this, examine the examples below:

```
PRINT 5>7                PRINT 5<7
Ø                        -1
PRINT 5<>7               PRINT 5<=7
-1                       -1
PRINT 5>=7               PRINT 5=7
Ø                        Ø
```

There will be many occasions when you'll need arithmetic expressions in your programs. If you follow these few simple rules for using numbers, the 99/4A will respond with speed and accuracy.

7 Variables



You can think of variables as a handy way to store information in the computer's memory. A variable's name identifies the memory location containing the information. The information may vary as a program runs. Computers recognize two types of variables, numeric and string. A numeric variable stores numbers; a string variable stores a series of letters, numbers, and symbols as a unit.

NUMERIC VARIABLES

The process of assigning a number to a letter in a program, is called "initializing the variable." You can use the LET statement to initialize a numeric variable.

```
100 LET A=99
```

Now, unless you change the value of A, the number 99 will be stored in a memory location called A. To demonstrate this, type in the following program. Remember to press ENTER after each program line and command.

```
100 LET A=99
110 PRINT A
120 A=A+1
130 PRINT A
140 END
RUN
```

The computer prints

```
 99

 100
```

because Line 100 assigns the number 99 to the variable A.

Line 110 instructs the computer to PRINT the value stored as A.
Line 120 increases the value of A by 1.
Line 130 tells the computer to PRINT the (new) value stored as A.

The little device in Line 120, A = A+1, lets you update variable values easily. The statement simply reassigns to the variable named A a new numeric value equal to the old A plus 1. To see this feature in action, type in the program below. (This program will continue to run until you break out by hitting FCTN:CLEAR.)

```
NEW
100 LET A=99
110 PRINT A::
120 A=A+2
130 PRINT A::
140 A=A-1
150 GOTO 110
RUN
```

Line 1ØØ assigns the number 99 to the variable A.

Line 11Ø prints the current value of A, then puts two blank lines in the display.

Line 12Ø increases the current value of A by 2.

Line 13Ø prints the current value of A, then skips the display down two more lines.

Line 14Ø decreases the current value of A by 1.

Line 15Ø sends the computer back to Line 11Ø, thus creating a continuous loop comprising Lines 11Ø through 15Ø inclusive.

Using the LET statement to assign values to variables is the common convention for most forms of BASIC. TI BASIC, however, allows you to initialize a variable without using LET. You can write Line 1ØØ as:

```
1ØØ A=99
```

and the program will still run properly. Since this form is simpler and more memory-efficient, we will use it from now on.

By the way, if you do not give a variable a value, the computer automatically assigns it a value of zero.

NAMING NUMERIC VARIABLES

You must follow a few simple rules when naming numeric variables.

All variable names must begin with a letter and may contain only letters and numbers. Thus, Z, Z5, L41P, FQR, COLORADO, PLAYER2, and PLAYERTWO are all legal variable names, but 5, 5Z, STREET#, @LP-5, A? are not.

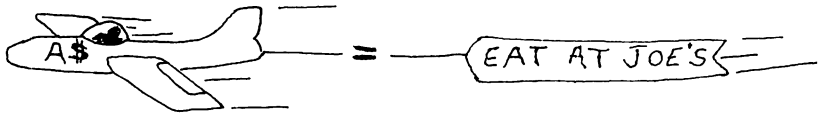
Different variables in the same program may not have the same name because the computer will interpret them as the same variable. Minor changes—one letter or number—are enough for the computer to distinguish variables.

It helps if you name your variables according to the part they play in your program. For example, DURATION is a good name for a variable that the controls the length of a particular event, such as a musical note or a graphics display.

A variable name must never exceed 15 letters. If it does, the computer responds with a BAD NAME error message.

One final rule about naming variables: You may not use BASIC reserved words as names. If you do, the computer will say CAN'T DO THAT. You may, however, include a reserved word *within* a name; thus RUN41 and HOMERUN are legal names.

STRING VARIABLES



String variables are used for characters (letters, numbers, and symbols) rather than numbers. The same rules apply to naming string variables as apply to numeric variables, with one exception: a string variable name always ends with a dollar sign ($). Thus A$, NAMES$ and Z52$ are all legal string variable names.

String variables differ from numeric variables in two other ways. First, numbers included in the string cannot perform or be used in mathematical operations. Second, all characters within a string must be enclosed in quotation marks. The following program shows how to use string variables.

```
NEW

100 A$="HERE ARE 3"

110 R43$="**EXAMPLES**OF**"

120 BOB$="STRING VARIABLES"

130 PRINT ::A$::R43$::BOB$

RUN
```

THE ROLE OF VARIABLES IN PROGRAMMING

The main purpose of variables is to help us use the computer to process information in the form of numbers or strings. These fundamental elements of information that are processed, and sometimes produced, by the computer are called DATA. One function at which your computer is great is organizing this collection of information, known as DATA, and presenting it in whatever order you have programmed. Your task as a programmer is to convert your information into DATA, and then structure it into a program that will present this DATA in the proper order at the proper time.

TI BASIC offers many tools with which to accomplish this task. Among them are a set of programming statements that allow you to

assign DATA to variables in a program. They include READ/DATA, INPUT, and FOR-NEXT loops. In addition, there are a set of **conditional statements** such as IF-THEN which are used to check the status of a variable. Their job is to check if a particular condition is true, and when it is, to direct the computer to perform another operation. A third set of statements known as **branching statements** allow you to transfer control from one line to another while a program is running. Branching statements such as GOTO and GOSUB are largely responsible for controlling the flow of the program.

In Chapter 9, "Branching Statements," we will explore some of the ways branching statements are used to control program flow. With this knowledge we can then explore more fully some of the many uses of variables and DATA assignment.

8 Using PRINT

PRINT is one of the most common and easy-to-use statements in the BASIC language. But it also has great potential for sophisticated applications. You'll find some of these applications illustrated in the demonstration programs in this book. Long after you have learned to program, you will still be discovering new ways to use PRINT.

THE PRINT SYSTEM

For printing purposes your computer's screen is divided into 28 vertical columns and 24 horizontal rows. The columns are numbered from left to right, the rows from top to bottom. Each horizontal line of 28 characters is further divided into two 14-character zones, left and right.

A screen of PRINTed text always appears at the bottom, with each new PRINT statement pushing any preceding material upward. This process may continue until previously PRINTed material is pushed off the top of the screen. The placement on the screen of material produced by each PRINT statement depends on the summed effect of all the PRINT statements that make up a particular screen display.

You may use a PRINT statement to print one item or several different items. The results you get depend on how you use four special punctuation marks: quotation marks and the three **print-separators**; colons, commas, and semicolons. Let's consider each of these in turn.

QUOTATION MARKS

Quotation marks are the most elementary PRINT punctuators. Because the computer "thinks" in numbers, any time you want it to deal with

non-numeric information you must tell it to take that information literally, not as a number. For instance, if you type PRINT BLUE, the computer will display a zero. This is because it interprets BLUE as the name of a number, whose value has not yet been assigned. Since the computer always assigns a value of zero to undefined numeric variables, it PRINTs a zero. If you want the computer to PRINT the word BLUE as is, then you must place it within quotation marks. Placing information within quotation marks creates a string, a list of characters which only represent themselves. Once you have made some piece of information into a string, you cannot freely perform manipulation of a numeric nature on it. (There is a highly structured set of rules in TI BASIC governing the relationship of strings and numbers primarily defined by the naming and use of string and numeric variables and functions.)

Quotation marks must always be used in pairs—one at the beginning of a string and one at the end. You may have more than one set of quotation marks in a given PRINT statement, but they must always be part of an opening and closing pair. Note also that blank spaces between quotation marks will be preserved in your string, since the computer considers them as characters.

If you wish to PRINT quotation marks by themselves, you must use a double set of quotation marks, two pairs to the left and two to the right. To print ", you must type PRINT """"; PRINT """ is incorrect. To print a *word* in quotes, you must use altogether six quotation marks. For instance, PRINT """"DOG"""" produces the result "DOG."

You can get around using quotation marks to print strings. For example:

```
100 A$="HOUSE"

110 A=12

120 PRINT A$

130 PRINT A
```

Line 100 defines the string variable A$ as "HOUSE."
Line 110 defines the numeric variable A as 12.
Line 120 displays the string HOUSE.
Line 130 displays the numeric value 12.

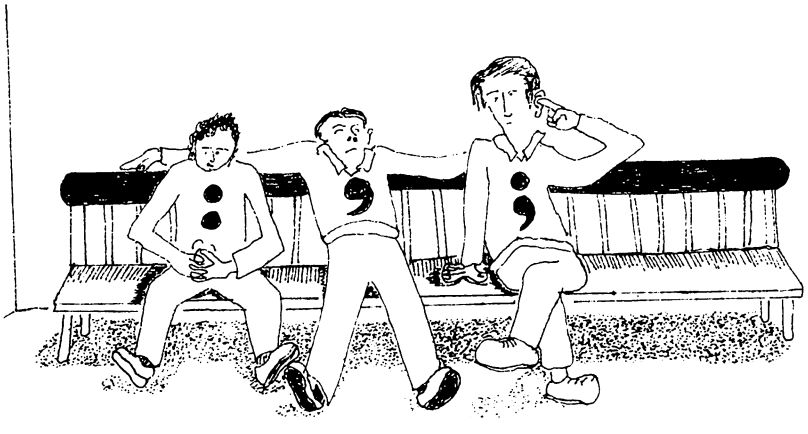
Thus, once you have created a string name you can print the string itself merely by asking the computer to print the string's name. Line 130 shows that the computer doesn't need quotation marks to print a number;

any numeric expression can be printed directly.

One more difference between numbers and strings appears in PRINTing. Numbers always leave an empty space before and after their location on the screen. These are called the **leading** and **trailing spaces.** In negative numbers, the minus sign occupies the leading space. Strings, on the other hand, have no leading and trailing spaces—so you can PRINT a series of strings side by side without gaps.

A special symbol, the ampersand (&), links strings together in a chain. If, for example, Line 12Ø in our previous program were changed to 12Ø PRINT A$ & A$, the computer would display HOUSEHOUSE. Such linking is called **concatenation.**

PRINT SEPARATORS



Three PRINT separators—the colon, comma, and semicolon—allow you to include more than one PRINT item in a program line. These print separators govern the spacing of PRINT items within a single print statement as well as PRINT items in different PRINT statements.

Colon (:). A colon places the next PRINT item at the beginning of the next line. When you use more than one in a single statement, the computer skips a line for every colon added to the first. Colons may be used as the first or last item of a PRINT statement or between items.

Comma (,). The comma places the next item in a PRINT statement in the next 14-character zone of the screen. If the preceding item is already located at the 15th column (where the right hand print zone begins), the next print item will appear at the beginning of the next row. Like colons, a number of commas may be strung together in a single PRINT statement. And, like colons, commas may be used before, after, or between items.

Semicolon (;). Semicolons place PRINTed items next to each other. For strings, there will be no space at all between items separated by semicolons. Numbers, however, will still carry leading and trailing spaces. Semicolons at the end of a PRINT statement can join items in succeeding PRINT statements, but only if there is room to place the additional print item(s) in the same screen row. If not, the additional item will PRINT at the beginning of the next row.

Unlike colons and commas, a number of semicolons strung together, will have the same effect as a single semicolon. There is no point to beginning a PRINT statement with a semicolon, since the new statement will automatically place its contents in a new row unless instructed by a semicolon at the end of the previous statement to do otherwise.

If a PRINT item needs to be set off by quotation marks, be sure to place the PRINT separators *outside* the quotation marks. Note, too, that the spacing between lines will not necessarily be the same if a print separator is used at the beginning of a second line as if used at the end of the first. This is because the computer automatically starts a new row for each new PRINT statement before evaluating the print separators within that line. Hence, print separators at the beginning of a line can be used to push previously PRINTed material upward on the screen, but not to bring it closer. To link items in different PRINT statements, you must always put the proper punctuation at the end of the previous statement.

This short program illustrates the points we've been discussing.

```
100 PRINT "THAT"::"WILL":"LA
ST":::
110 PRINT "HEY","JUMP","ALS
O",,,
120 PRINT 12;13;14
130 PRINT "COOK";
140 PRINT "BOOK"
```

Remember that you can also space items within a single PRINT statement by including empty spaces with the material within quotation marks. The computer will faithfully reproduce these spaces.

THE TAB FUNCTION

Used with PRINT statements, the TAB function gives you added control over the spacing of your material. TAB works much like the tabulator key on a typewriter—by putting material a specific number of spaces to the right. TAB is followed by a set of parentheses containing a number, numeric variable, or numeric expression. If needed, the computer rounds off this number to the nearest whole number; for instance, TAB(3.4) will move material three spaces to the right.

TAB works within the 28-column framework of the TI PRINT system. If you specify a TAB value greater than 28, the computer will subtract 28 from your value until it gets a number from 1 through 28. If your value is less than 1, the computer will make it into a 1.

The computer regards TAB as a PRINT item like any other in a PRINT statement; if you use more than one, you must separate them with colons, commas, or semicolons. TAB does not require quotation marks, however, and need not be preceded by a print separator if it is the first item in a PRINT statement or followed by a print separator if it is the last. Print separators affect TAB in the same manner as any other print item.

If you have text in a PRINT statement followed by a TAB, the material you want TABbed will only print on the same line as the preceding text if the number in the TAB function is greater than the number of spaces occupied by the previously printed text: the TABbed material will appear in the column whose value equals the value within the parentheses. If more spaces are occupied by previously printed text than given in the TAB function, the material will be TABbed to the same location in the next row. If the PRINT item to be TABbed cannot fit between the column specified in the TAB function and the end of that line, the material will not be TABbed.

Remember that the TAB function places material into the Nth column to the right of previously printed material, or from the beginning of a row. This is not the same as skipping N number of spaces. TAB(10) does not skip the first ten columns to place material into the eleventh, but actually places material in the tenth column.

The following program demonstrates the TAB function. The numbers in Line 15Ø end up in different rows, even though TAB(4) and TAB(7) leave spaces for the digits because not enough space is left for the number's trailing spaces.

```
100 PRINT TAB(4);"TABLE";TAB
(9);"TALK"
110 PRINT TAB(44);"HANDY"
120 PRINT TAB(45);"MAN"
130 PRINT TAB(5);"CONSERVATI
ON CREATES JOBS"
140 A=123
150 PRINT A;TAB(4);A;TAB(7);
A
160 PRINT TAB(4);"99/4A";TAB
(27);"TI"
170 A$="WORK"
180 B$="HORSE"
190 PRINT TAB(5);A$;TAB(9);B
$
```

THE SICK ROSE

Before leaving the PRINT statement, let's look at two programs that use different methods to format the same piece of text on the screen. Our text is a poem by William Blake from his *Songs of Experience*, entitled "The SICK ROSE." In the first program, we will stick to simple PRINT statements followed by quoted material. For spacing we will use additional PRINT statements to skip lines we wish to leave blank and empty spaces within quoted material.

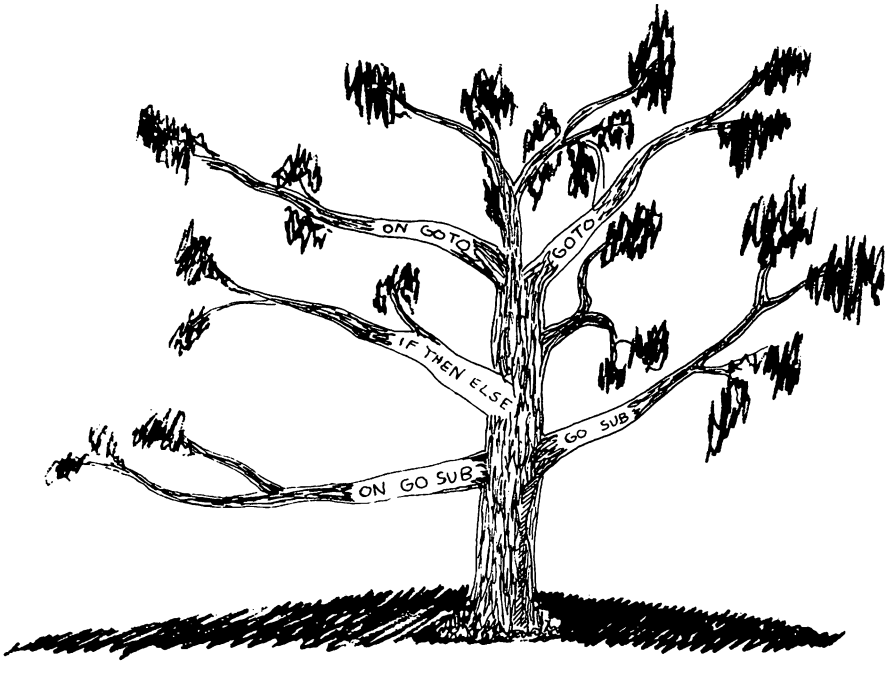
The second program is functionally identical to the first but uses print separators to add the empty lines. In addition, we will store the poem as a series of eight strings. All eight strings will be printed in a single PRINT statement at the end of the program, formatted by print separators. As you can see, print separators help create a more sophisticated, space-saving program.

```
100 CALL CLEAR
110 PRINT "        THE SICK
ROSE"
120 PRINT
130 PRINT "      BY WILLIAM
BLAKE"
140 PRINT
150 PRINT
160 PRINT "   O ROSE THOU AR
T SICK."
170 PRINT "   THE INVISIBLE
WORM,"
```

```
180 PRINT "   THAT FLIES IN
THE NIGHT"
190 PRINT "   IN THE HOWLING
 STORM:"
200 PRINT
210 PRINT "   HAS FOUND OUT
THY BED"
220 PRINT "   OF CRIMSON JOY
:"
230 PRINT "   AND HIS DARK S
ECRET LOVE"
240 PRINT "   DOES THY LIFE
DESTROY."
250 PRINT
260 PRINT
270 PRINT
280 GOTO 280
```

```
100 CALL CLEAR
110 PRINT TAB(9);"THE SICK R
OSE"::
120 PRINT TAB(7);"BY WILLIAM
 BLAKE:":::
130 A$="   O ROSE THOU ART S
ICK."
140 B$="   THE INVISIBLE WOR
M,"
150 C$="   THAT FLIES IN THE
 NIGHT"
160 D$="   IN THE HOWLING ST
ORM:"
170 E$="   HAS FOUND OUT THY
 BED"
180 F$="   OF CRIMSON JOY:"
190 G$="   AND HIS DARK SECR
ET LOVE"
200 H$="   DOES THY LIFE DES
TROY."

210 PRINT A$:B$:C$:D$::E$:F$
:G$:H$:::
220 GOTO 220
```

9 Branching Statements



Branching statements transfer action from one line or statement in a program to another by creating different "branches" along which program action may flow. These action directors come in two forms: unconditional statements that simply redirect program action; or conditional statements that allow two or more responses according to conditions set by the program's logic. Branching statements include GOTO, GOSUB, ON-GOTO, ON-GOSUB and IF-THEN-ELSE.

THE UNCONDITIONAL BRANCHERS

GOTO

This is the simplest, easiest to understand, and most popular branching command. It transfers program control from the numbered statement in which it appears to the numbered line specified in the GOTO statement. Hence, 100 GOTO 300 transfers program control from Line 100 to Line 300, skipping over the lines in between.

Let's go back to the "Diamond Track" program introduced in Chapter 5:

```
2 REM *** DIAMOND TRACK ***
3 CALL CLEAR
6 PRINT TAB(20);"LINE #"::
10 PRINT "          A
   10"
20 PRINT "         B B
   20"
30 PRINT "        C   C
   30"
40 PRINT "       D     D
   40"
50 PRINT "      E       E
   50"
60 PRINT "     F         F
   60"
70 PRINT "    G           G
   70"
80 PRINT "     H         H
   80"
90 PRINT "      I        I
   90"
100 PRINT "       J     J
   100"
110 PRINT "        K   K
   110"
120 PRINT "         L L
   120"
130 PRINT "          M
   130"
140 GOTO 10
```

One of the most useful aspects of the GOTO statement is shown graphically when this program is run: creation of a simple program loop. Sustaining program action by creating simple loops with GOTO can be done in two ways. Where it is desirable to repeat program action again and again, GOTO can be used to simply direct the program to restart the steps leading to the GOTO statement. In other cases, GOTO is used to occupy the computer so that some program effect can be observed at length. For instance:

```
500 CALL SCREEN(2)

RUN 500
```

turns the screen black—but only for a moment. If you add

```
510 GOTO 510
```

the screen stays black, because the computer has been tied up executing a one-line loop in Line 510. Creating simple loops with GOTO can come in handy when you want to see the effect of lines before you add them to a larger program.

Let's look at GOTO's effect by entering GOTO 110 in a (new) Line 45 in the Diamond Track program. The pattern now produces shrunken diamonds because Lines 50 to 100 are skipped over by the GOTO at Line 45. If we now insert a new line, 115 GOTO 50, the letters zig-zag back and forth without meeting in the middle (except at the beginning), because program action has now been caught in a loop between Lines 50 and 115; the GOTO statement in Line 45 has been excluded.

Now insert a new line, 105 GOTO 120. Once again, the GOTO at Line 105 intercepts the GOTO at Line 115, sending the program back to the beginning.

These examples illustrate the properties of unconditional branching statements. GOTO statements can either exclude or lead into one another, depending on how they are placed in a program. When designing a program, you must decide if you wish GOTO to introduce change or to preclude it.

GOSUBroutine

This unconditional branching statement implements program subroutines. A **subroutine** is a subsection within a program that performs a series of steps, or routine, that is called upon repeatedly as the program RUNs. The difference between GOTO and GOSUB is that GOSUB "remembers" where it came from and returns to the next line after the initial GOSUB statement. To ensure this, you must insert the statement RETURN as the last step in a subroutine. The basic structure of a subroutine with GOSUB and RETURN looks like this:

```
100 PROGRAM STATEMENT A
110 GOSUB 500
120 PROGRAM STATEMENT B
130 GOSUB 500
140 PROGRAM STATEMENT C
150 GOSUB 500
```

```
160 PROGRAM STATEMENT D
    |
    |
500 PROGRAM STATEMENT X
510 PROGRAM STATEMENT Y
520 PROGRAM STATEMENT Z
530 RETURN
```

First statement A is executed; then the program goes to Line 500, runs through the steps of the subroutine (Lines 500-530) and returns at Line 530 to program statement B. This pattern repeats each time there is another GOSUB statement.

Our next program demonstrates GOSUB in action. The first set of statements place vertical columns characters on the screen. After each of these statements has been performed, the subroutine starting at Line 240 is run; the program then RETURNs to the next call for characters until it reaches the END in Line 230. The subroutine produces our sound effects.

GOSUB Demo

```
100 CALL CLEAR
110 CALL VCHAR(1,11,71,24)
120 GOSUB 240
130 CALL VCHAR(1,13,79,24)
140 GOSUB 240
150 CALL VCHAR(1,15,83,24)
160 GOSUB 240
170 CALL VCHAR(1,17,85,24)
180 GOSUB 240
190 CALL VCHAR(1,19,66,24)
```

```
200 GOSUB 240
210 CALL VCHAR(1,21,83,24)
220 GOSUB 240
230 END
240 CALL SOUND(200,-2,0)
250 CALL SOUND(300,-3,0)
260 CALL SOUND(280,-5,0)
270 CALL SOUND (440,-1,0)
280 RETURN
```

also be true. Multiplication is the **logical AND operator** because both expressions, A *AND* B, must be true, or the result is judged false.

Whenever the overall result of the expression between IF and THEN is true, the computer moves to the line number immediately after THEN; if false, it moves to the lines number given after ELSE. You need not specify ELSE, however; the computer will proceed to the next line program statement automatically.

The IF-THEN statement is an incredibly flexible programming tool. You can set up many essential shifts in program flow with IF-THEN statements. You can also set limits to changes in variable values—limits that can prevent program malfunctions. For example:

You can even call up another subroutine from a subroutine. Add the following lines to our GOSUB Demo program:

```
255 GOSUB 300          300 CALL SCREEN(16)
    |                  310 CALL SCREEN(14)
    |                  320 RETURN
```

The new lines interrupt the sound subroutine each time it RUNs and call a new subroutine that changes the screen color, first to white, then to magenta.

CONDITIONAL BRANCHING STATEMENTS

Let's now turn to branching statements that actually select among alternative program steps according to conditions set by program logic. ON-GOTO, ON-GOSUB, and IF-THEN-ELSE statements can all recognize program logic in different ways and with different effects. Using these statements well is crucial to developing reasonably complex programs.

ON-GOTO

This statement uses either the value of a numeric expression or variable to determine which program statement to GOTO. The numeric expression or the variable's name belongs between ON and GOTO in an ON-GOTO statement.

The computer first finds the value of the numeric expression and rounds it off to the next integer. This integer is then used as a pointer

which chooses among numbered program lines. If, for example, the value reached is 1, the computer proceeds to the *first* line number given in the ON-GOTO statement; if the value is 4, the computer moves to the fourth line number in the ON-GOTO statement. For instance,

```
100 CALL CLEAR
110 X=3
120 Y=2
130 ON (X/Y)*2 GOTO 140,150,
160
160 PRINT "WOW"
```

prints "WOW" because 3 divided by 2 times 2 equals 3. Since 160 is the third line number following the ON-GOTO statement in Line 130, the computer goes to Line 160 and prints WOW. If the expression in the ON-GOTO statement equals either zero or a number higher than the number of lines listed in the statement, you will get an error message.

```
100 REM ***ON-GOTO DEMO***
110 CALL CLEAR
120 PRINT "BATTER'S UP!! HOW
 MANY BASES WILL YOU REACH?(
1 TO 4)"::
130 INPUT BASES
140 ON BASES GOTO 150,170,19
0,210
150 PRINT "YOU'LL GET A SING
LE!"::
160 GOTO 120
170 PRINT "YOU'LL GET A DOUB
LE!!"::
180 GOTO 120
190 PRINT "YOU'LL GET A TRIP
LE!!!"::
200 GOTO 120
210 PRINT "HOME RUN HITTER!!
!!"::
220 GOTO 120
```

ON-GOSUB

This statement works much like ON-GOTO except that the computer returns to the line immediately after an ON-GOSUB statement when it encounters a RETURN statement at the end of a given subroutine. The following program uses three subroutines to calculate your age in months, days, or hours. Based on the value of the input variable,

CHOICE, the ON-GOSUB in Line 180 directs the computer to the appropriate subroutine. When the computer RETURNs from each subroutine, it executes Line 190 GOTO 100, and the program starts over. Since there are only three different subroutines, CHOICE can take a value from 1 to 3 only; any other number will produce an error message and interrupt the program.

```
100 REM ***ON-GOSUB DEMO***
110 CALL CLEAR
120 PRINT "HOW OLD WILL YOU
BE ON YOUR NEXT BIRTHDAY?"::
130 INPUT AGE
140 PRINT "TO FIND YOUR AGE
IN....":::
150 PRINT "MONTHS, PRESS (1)
"::
160 PRINT "DAYS, PRESS (2)":
:
170 PRINT "HOURS PRESS (3)":
:
180 INPUT CHOICE
190 ON CHOICE GOSUB 210,240,
270
200 GOTO 120
210 MONTHS=AGE*12
220 PRINT "YOU WILL BE";MONT
HS;"MONTHS OLD"::
230 RETURN
240 DAYS=AGE*365
250 PRINT "YOU WILL BE";DAYS
;"DAYS OLD"::
260 RETURN
270 HOURS=AGE*8760
280 PRINT "YOU WILL BE";HOUR
S;"HOURS OLD"::
290 RETURN
```

IF-THEN-ELSE

This conditional branching statement gives the computer a choice of two different program steps or subroutines based on the evaluation of either a numeric expression and a relational expression. Conceptually, the IF-THEN-ELSE statement looks like this:

IF (A is true) THEN

[line number] ELSE (A is false) [alternate line number]

where A is either a numeric expression or a statement of relationships between two or more factors. For example, we might write:

```
IF A<=16 THEN 100 ELSE 200
```

meaning that if the variable A is 16 or less, program control should go to Line 100; otherwise, it should go to Line 200. The "GOTO" function of the IF-THEN-ELSE statement is automatic.

The subtlety of the IF-THEN-ELSE statement comes into play when two or more expressions are combined into a single true or false result. Each expression in an IF-THEN-ELSE statement is evaluated by the computer and assigned a value. If false, the value assigned is a zero. If true, the value is less than or greater than zero. The values assigned to each expression in an IF-THEN-ELSE statement can either be added, or multiplied by, each other to produce an overall true/false value for the entire statement.

The logic table below lists all the possible combinations of two expressions (A and B) when they are either added or multiplied to yield one overall true/false value.

TRUTH TABLE FOR LOGICAL AND AND OR

	Both False	One True/One False	Both True
	(A) (B)	(A) (B)	(A) (B)
ADDED:	(0) + (0) = 0(False)	(−1) + (0) = −1(True)	(−1) + (−1) = −2(True)
MULTIPLIED	(0) * (0) = 0(False)	(−1) * (0) = 0 (False)	(−1) * (−1) = 1 (True)

As you can see, addition and multiplication yield the same results when both A and B are either true or false. When one expression is true and one false, however, multiplication makes the final result false, while addition makes it true. Addition is therefore termed the **logical OR operator.** If either one expression *OR* the other is true, the combined results will will quickly produce BAD VALUE IN 130 because the computer does not recognize any valid value for screen color above 16. You can remedy this problem easily by adding the line

```
135 IF X>15 THEN 110
```

which returns the value of the screen color to 1 as soon as X becomes greater than 15. (There is no point in using ELSE 140 here because the

program naturally proceeds to Line 140 as long as X>15 is false.)

IF-THEN Music

The following program will show you the kind of sophisticated control you can have over program flow and changing variable values with IF-THEN statements. The "IF-THEN Music" program merely moves the frequency of a tone up and down between 120 cycles and 7500 cycles per second.

The program is divided into two parts, one devoted to raising the pitch in the CALL SOUND statement on Line 150, and the other to bringing the pitch back down in the CALL SOUND statement on Line 210. There are four IF-THEN statements on Lines 140, 160, 220, and 240. Those on Lines 140 and 240 keep the sound frequency variable, X, between 120 and 7500 cycles by shifting program control between the rising and falling parts of the program, thus changing the tone's direction. When X gets low or high enough, these IF-THENs become true statements, and program flow shifts.

The two IF-THEN statements at Lines 160 and 220 are a little trickier. They set conditions under which the frequency being used in the CALL SOUND statement will or will not be printed. Notice that Line 160 uses the logical connector of addition, Line 220 the logical connector of multiplication. Line 160 prints all frequency values less than 150 OR greater than 6500. Line 220 sets up the conditions that direct the computer to print only those values greater than 1500 AND less than 2000. Rising X values are printed on the left side of the screen and the falling values on the right.

Switch the "+" in Line 160 to "*", and the "*" in Line 220 to "+". Check the results with the "Truth Table."

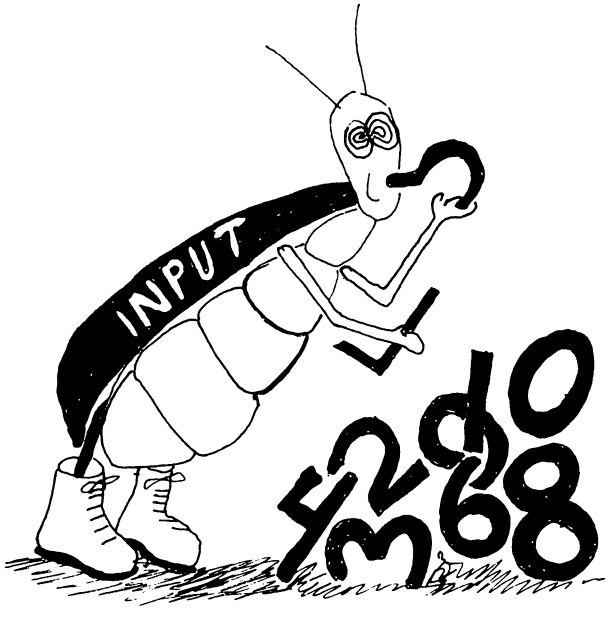
```
100 REM ***IF-THEN MUSIC***
110 CALL CLEAR
120 X=120
130 X=X*1.05
140 IF X>7500 THEN 190
150 CALL SOUND(70,X,0)
160 IF (X<150)+(X>6500)THEN
170 ELSE 130
170 PRINT X
180 GOTO 130
190 X=7500
200 X=X/1.05
210 CALL SOUND(70,X,0)
220 IF (X>1500)*(X<2000)THEN
```

```
230 ELSE 240
230 PRINT, X
240 IF X<120 THEN 120 ELSE 2
00
```

10 Data Anyone?

There are a number of ways by which we may enter DATA into a program and manipulate it with the help of variables and branching statements. So far, we've entered data, or defined variables in three ways: the LET statement (LET A = 3), the simplified version (A = 3), and our updating trick (A = A + 1). We can also use an INPUT statement.

INPUT



The INPUT statement allows you to enter values from the keyboard while a program is running; the program pauses while you enter the data.

We'll use the simplest form of INPUT in the next program. When you RUN the program a question mark appears. Type in a word or name and press ENTER. The computer then PRINTs your word or name, showing that the word has been stored as the value of the string variable B$. Another question mark appears. This time, type in a number and press ENTER. The computer PRINTs the number, which it has assigned to the numeric variable C.

```
100 INPUT B$
110 PRINT B$
120 INPUT C
130 PRINT C
RUN
? SPOCK              [YOU TYPE SPOCK]
SPOCK                [THE COMPUTER PRINTS]
? 42                 [YOU TYPE 42]
 42                  [THE COMPUTER PRINTS]
```

Data entered after an INPUT prompt (?) must be valid for the computer to accept it. For instance, if you tried to enter a string in response to the second INPUT prompt, you would get the following error messages:

```
* WARNING:
  INPUT ERROR IN 120
TRY AGAIN:
```

This is because Line 120 is asking for a numeric value.

To avoid this problem you can write your own INPUT prompts. Besides promoting INPUT accuracy, personalized prompts make your programs truly interactive. You do need to follow two simple rules: enclose your prompt text in quotation marks and follow it with a colon.

Here's a program which is similar in form to our earlier INPUT program. The main difference is the prompt. We've added the CALL

CLEAR and PRINT statements for effect. Make sure all your punctuation is correct. Also, leave blank spaces where we have.

```
100 CALL CLEAR
110 INPUT "WHAT IS YOUR NAME
?":B$
120 PRINT ::"VERY WELL, ";B$
130 INPUT "HOW MANY LANGUAGE
S DO YOU SPEAK?":C
140 PRINT ::C;"IS VERY IMPRE
SSIVE "B$
RUN

WHAT IS YOUR NAME? SPOCK          [RESPONSE]

VERY WELL, SPOCK                  [PRINTS]
HOW MANY LANGUAGES DO YOU SP      [PROMPT]
EAK? 42                           [RESPONSE]

42 IS VERY IMPRESSIVE SPOCK       [PRINTS]
** DONE **
```

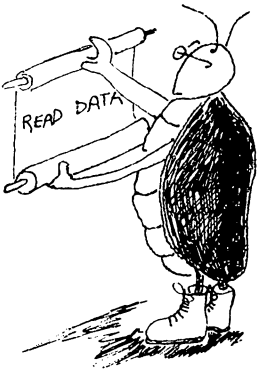
You can also combine INPUT with a PRINT statement, giving you the ease of a simple INPUT statement plus the clarity of a prompt.

```
100 CALL CLEAR
110 PRINT "HOW OLD ARE YOU?"
120 INPUT A
130 PRINT "OH, YOU WERE BORN
 IN"
140 PRINT 1983-A
RUN
```

```
HOW OLD ARE YOU                [PRINTS]
? 31                           [RESPONSE]
OH, YOU WERE BORN IN           [PRINTS]
1952                           [PRINTS]
```

Now that we have added INPUT to our list of statements for assigning values to variables, let's explore another TI BASIC statement that does this in yet another way.

THE READ/DATA STATEMENT



One of the most efficient methods for entering and controlling both large and small amounts of data in a program is the READ/DATA statement. The READ/DATA statement comprises two parts. DATA represents the information you want to enter into the program; this information goes into a section of the program called the **DATA bank,** or DATA list. The READ portion of the statement extracts information from the DATA bank and assigns it to a variable.

Let's take a look at the READ/DATA statement in action. Type the program below. When you RUN it, the computer will PRINT a list of famous people born on November 30. It will also give a *DATA ERROR IN 13Ø message. Don't worry; we'll explain why.

```
100 REM *** STEP ONE ***
110 CALL CLEAR
120 PRINT "FAMOUS PEOPLE BOR
N ON NOV.30"::
130 READ A$
140 PRINT A$
150 GOTO 130
500 DATA JOHN BUNYAN,MARK TW
AIN, WINSTON CHURCHILL
510 DATA SHIRLEY CHISHOLM,AB
BIE HOFFMAN
```

If your program ran correctly, your screen cleared and the computer printed FAMOUS PEOPLE BORN ON NOV. 30 followed by the names contained in the DATA bank. If not, go back and check your work for errors. Are all the commas in the right places? They are very important. Spaces after the commas are, however, ignored.

Now, let's take a closer look.

Line 110 clears the screen.

Line 120 prints the program title; the two colons tell the computer to skip two rows.

Line 130 instructs the computer to READ the first piece of DATA in the DATA bank and to name it A$.

Line 140 prints the current value of A$.

Line 150 creates a loop, sending the computer back to Line 120 to READ the next piece of DATA.

Lines 500-510 contain the DATA.

Whenever the 99/4A encounters a READ statement, it will immediately READ a piece of DATA. In this case, we instructed it to read only one piece of DATA and to assign it to the variable A$. The computer will always READ DATA from left to right and from the lowest line number to the highest; a **DATA pointer** keeps track of the computer's place in the DATA bank. The individual DATA values are always

separated by commas. The last piece of DATA on any given line, however, should *not* be followed by a comma. (We'll explain why in the next section.)

Placing the READ statement in a loop, as we did, enters the DATA into our program piece by piece. But our program has no way to break out of the loop once the computer has READ the DATA. That's why we got the message DATA ERROR IN 13Ø. The 99/4A READ the last piece of DATA, "ABBIE HOFFMAN", assigned it to A$, printed ABBIE HOFFMAN(A$), and was then sent back to Line 13Ø to READ again. But this time through the loop, it found nothing in the DATA bank and so printed the error message.

Can you think of a way to get out of the loop? Hint: try an IF-THEN statement. (We'll tell you one possible solution after discussing how to set up a DATA bank.)

SETTING UP A DATA BANK

A DATA bank may appear anywhere in your program. We usually place it at the end leaving plenty of space for future line additions. Some programmers prefer to place their DATA statements toward the beginning; others put them near the READ statements. The choice is yours.

Although the DATA bank may go anywhere in your program, the DATA must be written in the exact order that you want it READ into the program. Remember: the computer READs DATA from left to right and from the lowest line number to the highest.

There are a few simple rules to follow in setting up your DATA bank. First, choose a line number—in our case, Line 5ØØ. Then type in a space and the word DATA followed by another space. You can now begin typing in DATA values, but be sure that these values match the type of variable you're assigning them to. Since we are using names in our program, we need a string variable such as A$. For numeric DATA, use a numeric variable name.

As mentioned before, you must put a comma after each piece of DATA except the last one. For numeric DATA, a comma after the last entry on a given line throws the computer out of sync and causes an error message. Placing a comma at the end of a DATA line containing strings will print a blank line. For example, if you add a comma after WINSTON CHURCHILL in Line 5ØØ, you would get a display with a blank line between WINSTON CHURCHILL and SHIRLEY CHISHOLM. This is because the computer assumes that you want to enter a null string (one with no characters). Similarly, two commas placed one after the other in a string DATA statement will be READ as a null string. Two commas in a numeric DATA, in contrast, will produce an error message (e.g., DATA ERROR IN 13Ø).

If you wish to include a comma as *part* of a string, you must enclose the whole string in quotes. This rule also holds for leading and trailing spaces. Thus MAYBE, TOMORROW would appear in a DATA list as follows.

```
5ØØ DATA "MAYBE, TOMORROW"
```

Your 99/4A will accept up to four screen rows of DATA per line number, but it's best to use only one or two. This makes it quicker and easier to spot and correct errors.

One final rule about setting up your DATA bank: always include the same number of DATA values as variable names in the READ statement; otherwise, the computer will look for a non-existent value. For example, if you have a READ A,B statement in your program, you could not have a DATA statement like DATA 1,2,3. This would cause a DATA ERROR message.

CHECKING VARIABLE STATUS

Now let's GET BACK to breaking out of the loop between Lines

13Ø-15Ø. You'll recall that we suggested using the IF-THEN statement: IF a particular condition is true, THEN the computer moves to another numbered line.

Let's insert an IF-THEN statement into our program on Line 145, right after the PRINT A$ statement, and use it to check whether the computer has READ the last DATA item (ABBIE HOFFMAN). IF it has, THEN we'll tell it to go to Line 4ØØ, where it will find an instruction to END the program. To do this, type and ENTER the two lines below.

```
145 IF A$="ABBIE HOFFMAN" TH
EN 4ØØ
4ØØ END
```

Notice that we used quotation marks around the string in Line 145. Although we are allowed to forego quotation marks in string DATA statements, we must, according to the rules of TI BASIC, use them around string values mentioned outside of the bank.

Now, RUN your program with the additions; if all goes well you should see a cheery ** DONE **.

MULTIPLE-VARIABLE READ/DATA STATEMENTS

You are not limited to READing only one variable at a time; in fact, you can READ several lines at once (although anything more than about seven or eight gets tricky). To see how multiple-variable assignment works, let's add another set of information to our "Famous People" program: the year each person was born. We'll need several changes, so you can exercise your editing skills. The complete program appears below.

```
1ØØ REM ***STEP TWO***
11Ø CALL CLEAR
12Ø PRINT "FAMOUS PEOPLE BOR
N ON NOV.3Ø"::
13Ø READ A$,YEAR
14Ø PRINT A$;TAB(18);YEAR
145 IF YEAR=1936 THEN 4ØØ
15Ø GOTO 13Ø
4ØØ END
5ØØ DATA JOHN BUNYAN,1628,MA
RK TWAIN,1835,WINSTON CHURCH
ILL,1874
51Ø DATA SHIRLEY CHISHOLM,19
24,ABBIE HOFFMAN,1936
```

Aside from the slight change to the REM statement in Line 1ØØ, our first real change occurs in the READ statement, Line 13Ø:

```
13Ø READ A$,YEAR
```

This demonstrates the correct form for a multiple-variable READ statement. Here we have added the numeric variable name YEAR; a comma separates it from the preceding variable A$. *Always place a comma between variables in a READ statement:* READ A,B,D$,C is correct; READ A B D$ C is not. With a second variable in the READ statement, the computer will now READ two pieces of DATA at a time.

The next change occurs in Line 14Ø. Here we add the TAB function to place the YEAR in column 18, thus separating it from the name (A$).

Line 145 contains an important change to the IF-THEN statement. Since this line checks if the last DATA item has been READ, we must adjust it to reflect our change in variables. Thus, A$ must be replaced by YEAR, and "ABBIE HOFFMAN" must be replaced by 1936.

The final changes occur in the DATA statements. Here, we simply INSERT the information for YEAR after the comma following each name. Note that a comma must be added after each new DATA entry.

With these changes, our program should flow properly and print out our famous people and the year they were born. You will find the READ/DATA statement a valuable addition to your growing programming repertoire. A statement that goes hand-in-hand with the READ/DATA statement is the RESTORE statement.

RESTORE

Once the DATA pointer has reached the final piece of information in the DATA bank, the computer can READ no more and prints an error message. At this point you must decide whether to break out of the READ loop and go on to another part of your program, or to exit the loop and END the program, or to reactivate the DATA with a RESTORE statement.

RESTORE instructs the computer to reuse the DATA by resetting the pointer to the beginning of the DATA, or, if you wish, to the beginning of any DATA line. The computer then begins READing the DATA over again, repeating a specific READ/DATA section of your program.

Here is a sample graphics program demonstrating the RESTORE statement. Type it in and RUN it; since the program runs continuously, you'll have to press FCT:CLEAR to break out.

```
100 CALL CLEAR
110 CALL SCREEN(16)
120 READ STAR$
130 PRINT TAB(12);STAR$
140 IF STAR$="*********" THE
N 160
150 GOTO 120
160 RESTORE
170 GOTO 120
300 DATA *,**,***,****
310 DATA *****,******
320 DATA *******,********
330 DATA *********
```

When the program is RUN your screen should turn white and clear. The pattern below should then scroll up your screen.

```
*
**
***
****
*****
******
*******
********
*********
```

If your program does not RUN correctly, check for errors in typing, spacing, punctuation, and the number and placement of stars in the data bank.

Here is a line-by-line description of how this program works.

Line 1ØØ	clears the screen.
Line 11Ø	turns the screen color white.
Line 12Ø	READs the first piece of DATA and assigns it to the variable named STAR$.
Line 13Ø	indents 12 spaces and PRINTs the current value of STAR$.
Line 14Ø	tests to see IF STAR$ is equal to the last piece of DATA (nine *s). IF it is, THEN the computer moves to Line 16Ø where it will receive an instruction to RESTORE the DATA.
Line 15Ø	creates the READ loop by sending the computer back to 12Ø to READ the next piece of information.
Line 16Ø	RESTOREs the DATA by setting the DATA pointer back to the first piece of information in the DATA bank.
Line 17Ø	creates another READ loop by sending the computer back to 12Ø after the DATA has been RESTORED.
Lines 3ØØ-33Ø	contain the DATA bank.

This program format is very similar to the "Famous People" program. The real difference occurs in Line 16Ø, which RESTORES the DATA, and Line 17Ø, which causes the program to RUN in a continuous loop.

When the RESTORE statement stands alone, all the DATA is RESTOREd. If you wish to RESTORE only part of it, you may add a line number after the RESTORE statement. This moves the DATA pointer to the first DATA item in that line and starts the READing from there. To see how this works type in this simple change.

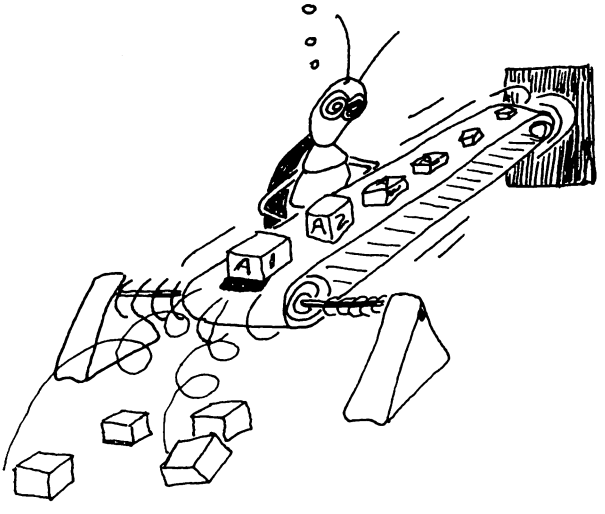
```
16Ø RESTORE 31Ø
```

Now RUN your program for an entirely different pattern. Note that the first pass through this program is identical to the original pattern because the computer first uses all the DATA. Only after the DATA is RESTOREd does the change appear.

RESTORE comes in handy in many program situations. To use it effectively, you should learn two more DATA-manipulating devices: the **counter** (A = A + 1), which you have already used to update variables, and DATA **flags** (also known as delimitors).

Let's look at counters first, as they are especially helpful when used with RESTORE.

USING COUNTERS TO MANIPULATE DATA



In our last program the RESTORE statement was part of a continuous loop; we had to press FCTN:CLEAR to stop the pattern. If you want to specify the number of repetitions, you need a counter and an IF-THEN statement.

Type in the following changes, and your star program will RUN three times:

```
160 RESTORE
170 A=A+1
180 IF A=3 THEN 200
190 GOTO 120
200 END
```

Your program should now look like this:

```
100 CALL CLEAR
110 CALL SCREEN(16)
120 READ STAR$
130 PRINT TAB(12);STAR$
140 IF STAR$="**********" THE
N 160
150 GOTO 120
160 RESTORE
170 A=A+1
180 IF A=3 THEN 200
190 GOTO 120
200 END
300 DATA *,**,***,****
310 DATA *****,******
320 DATA *******,********
330 DATA *********
```

The key to controlling the number of repetitions lies in Lines 160-200.

Line 160 RESTOREs the DATA after the first pass.

Line 170 sets up a counter. Since A was not initialized (assigned a value) earlier, it is automatically set at zero. The first time the computer passes the counter it registers 1 (0 + 1 = 1). The second time it registers 2 (1 + 1 = 2), and the third time through, A becomes 3 (2 + 1 = 3).

Line 180 tests IF A = 3. IF it does, THEN the computer goes to Line 200, where it finds an instruction to END the program, thus breaking the loop.

Line 190 sends the computer back to Line 120 to start READing DATA once again. This statement is bypassed when A = 3. This is why you must place the counter and the IF-THEN statement *before* the looping GOTO statement in Line 190.

Line 200 ENDs the program after three repetitions.

Counters work efficiently in a variety of situations. The next section will add another programming device counter to your bag of tricks.

USING DATA FLAGS

A DATA flag is an item in a DATA bank which acts as a marker that can be tested by an IF-THEN statement. IF the flag is READ, THEN the computer will exit the READ/DATA loop; you can then re-enter the loop or go on to something else.

Let's use flags to change our stars again. Make these additions and changes to your program:

```
125 IF STAR$="-1" THEN 250

180 IF A=2 THEN 200

250 Z=Z+1

260 IF Z=2 THEN 400

270 GOTO 120

310 DATA *****,-1,******

400 RESTORE

410 GOTO 120
```

This time, we're are not going to give you the complete program listing. Instead, treat this as an editing exercise. To check your work, RUN the program and compare your display with the one below.

```
*
**
***
****
*****
******
*******
********
*********
*
**
***
****
*****
*
**
***
****
*****
******
*******
********
*********
```

Note that the top three lines of stars scroll off the top of your screen.

How do these changes affect the program?

Line 31Ø places a −1 in the DATA. The −1 is the flag.

Line 125 checks the value of the variable STAR$. IF it equals −1, THEN the computer passes to Line 25Ø. IF it does not equals −1, THEN the PRINT statement is executed. Thus, we prevent the flag from being printed in the program display.

Line 25Ø is executed only when the computer READs the −1 flag. The counter set up here keeps track of how many times the flag has been READ.

Line 26Ø checks the flag counter. IF the counter equals 2, THEN control passes to Line 4ØØ. IF the counter equals any number other than 2, THEN the GOTO 12Ø statement of the next line (27Ø) is executed.

Line 4ØØ RESTOREs the DATA from the flag back to the DATA bank.

Line 41Ø loops back to the READ statement.

The first time the computer encounters the flag, it branches off to Line 25Ø where it increments the counter. It then checks in Line 26Ø IF Z = 2. Since Z = 1, the GOTO statement of Line 27Ø sends control back to the READ statement, where the rest of the DATA is READ and PRINTed. This makes the first full pattern.

The second time the flag is passed, the flag counter Z becomes 2, and the IF Z = 2 THEN 4ØØ statement applies. The DATA is RESTOREd and control passes once again to the READ STAR$ statement. In this way the second, shorter pattern is generated.

On the final pass, Z = 3, which has no effect on PRINTing the DATA; thus, a full pattern is created. Since a RESTORE statement was added to the program in Line 4ØØ, we bypassed the RESTORE in Line 16Ø by changing A from 3 to 2 in Line 18Ø.

To see just how and where the flag counter works, add this line:

```
255 PRINT Z
```

One final note on flags: always include as many flags in a DATA statement as you have variables in the READ statement . Thus, READ A,B,B2 requires −1, −1, −1 added to the DATA. If you are short one flag, the computer will READ some of your DATA as its flag, and that DATA will no longer be available for use in the program. In most cases, a DATA error message will appear somewhere down the line.

Now that you have a handle on the READ/DATA statement, let's turn to another important programming statement, FOR-NEXT.

11 The FOR-NEXT Loop

In our last chapter, we learned how to enter data into a program by placing the READ/DATA statement within a loop. We also learned how to control the loop through the use of counters and the IF-THEN statement. The FOR-NEXT loop is another method that will enable you to form loops and assign values to variables with fewer statements and greater control than the previous method. This is because the FOR and NEXT statements allow for a more concise definition of a loop.

Here is an example of a simple FOR-NEXT loop.

```
100 FOR X=1 TO 5
110 PRINT X
120 NEXT X
RUN
```

```
 1
 2
 3
 4
 5
** DONE **
```

Line 100 tells the computer how many times to runs through the loop. The X in this statement is the **loop variable**: 1 is its initial value and 5 is its terminal value.

Line 110 PRINTS the current value of X.

Line 120 instructs the computer to return FOR the NEXT X.

The FOR and NEXT statements together perform the same function as separate GOTO and IF-THEN statements. The loop variable starts at the initial value, in this case, 1. It is then tested against the terminal value (5) to see if the loop should continue. Since 1 is less than 5, con-

trol then passes to the next line, 11Ø. Since 1 is the current value of X, the computer will PRINT 1 and then pass control on to Line 12Ø. The NEXT statement of Line 12Ø then increments the loop variable by 1 and transfers control back to the line with the FOR statement, thus completing the loop. This loop will cycle until the NEXT statement increments the loop variable to 6 and transfers control back to the FOR X = 1 TO 5. At this point, the computer tests the value of X (6) against the terminal value of X (5). Since X now exceeds the terminal value, the computer breaks out of the loop, transferring control to the line immediately following the NEXT statement. In this case, no Line 13Ø exists, so the computer PRINTs "DONE."

For a loop to work properly, a FOR statement must always be accompanied by a NEXT. If you forget, the computer will return a FOR-NEXT ERROR message. You must also include the variable name mentioned in the FOR statement in the NEXT statement.

To see that the value of X indeed reaches 6 and that control then passes to the line after NEXT X, add this line to your program:

```
130 PRINT "X="; X
```

ENTERING DATA WITH FOR-NEXT LOOPS

We have seen how to use READ/DATA statements to enter DATA into a program. Now let's try a FOR-NEXT loop.

In this program, we will enter five different values for the variable X and multiply each value by 5.

```
100 CALL CLEAR
110 FOR X=1 TO 5
120 PRINT 5*X
130 NEXT X
RUN
```

```
 10
 15
 20
 25
** DONE **
```

5

To get the same results with READ/DATA statements, you would have to write:

```
100 CALL CLEAR
110 READ A
120 PRINT A*5
130 IF A=5 THEN 150
140 GOTO 110
150 END
200 DATA 1,2,3,4,5
```

The FOR-NEXT format is more concise in this case, but not always. For each program you will have to assess the nature and the amount of DATA you wish to process and choose your format accordingly. That's the beauty of mastering several programming options. READ/DATA and FOR-NEXT loops are just two of the BASIC ways to INPUT your DATA; we'll discuss another format, called an array, in Chapter 17.

DEFINING LOOPS WITH STEP

We've seen that the FOR statement defines your loops, but you are not limited to simple definitions like FOR X = 1 TO 5. You may, for instance, set the initial value to something other than 1. Thus, FOR X = 5 TO 28 would be a legal statement defining a loop that begins at 5 and ends at 28. This option is particularly apt when you want to enter DATA that must fall within certain numerical limits.

The word STEP added to a FOR statement further defines a loop. STEP increments the loop variable values by any number you insert after the STEP expression. For example:

```
100 CALL CLEAR
110 FOR I=5 TO 25 STEP 5
120 PRINT I
130 NEXT I
RUN
```

```
 5
 10
 15
 20
 25
```

You may also use STEP with a negative number to enter your DATA in descending order. If you do, remember to reverse the written order of the initial and terminal loop variables values. FOR I = 5 TO 25 STEP −5 is thus an invalid statement, but FOR I = 25 TO 5 STEP −5 is valid. To see the negative STEP statement at work, change Line 110 of the last program to:

```
110 FOR I=25 TO 5 STEP -5
```

You should get

```
25
20
15
```

```
 10
 5
```

Although you don't need the STEP statement to increment a variable by 1, you do need it for −1.

Here is a graphics program with simple and STEPped FOR-NEXT loops. The loops enter DATA into a CALL HCHAR statement to create a "Criss Cross."

As you will recall, CALL HCHAR controls the horizontal graphics display. The format for this statement is CALL HCHAR(x,y,z) where x equals the row number, y equals the column number and z equals the character code number. We will use the character code 42 for the asterisk. The variable X is used along with a counter to control the row number, and the variable Y is used with the FOR-NEXT loop to control the column number.

The program runs in a continuous loop, so you must use FCTN:CLEAR to break out of the cycle.

```
100 REM ***CRISS CROSS***
110 CALL CLEAR
120 FOR Y=5 TO 28
130 X=X+1
140 CALL HCHAR(X,Y,42)
150 NEXT Y
160 X=0
170 FOR Y=28 TO 5 STEP -1
180 X=X+1
190 CALL HCHAR(X,Y,42)
200 NEXT Y
210 GOTO 210
RUN
```

Did you get a diagonal line of asterisks from the top left corner of the screen to the bottom right corner? And a second diagonal from the upper right to the lower left? Good.

Line by line:

Line 1ØØ is the title.

Line 11Ø clears the screen.

Line 12Ø defines a loop that assigns the values 5 through 28 to the variable Y (column numbers).

Line 13Ø sets up a counter to generate row numbers, X. Each time through the loop, the counter increments by 1. Since the loop has 24 numbers, the counter generates the numbers 1 to 24.

Line 14Ø CALLs the horizontal character display. The first time through the loop, X = 1 and Y = 5. The second time, X = 2 and Y = 6. This continues until X = 24 and Y = 28.

Line 15Ø contains the NEXT statement to go with the FOR statement in Line 12Ø. It instructs the computer to increment the variable Y and to return to the FOR statement.

Line 16Ø reinitializes the row counter X to 0. This allows us to re-use the counter in the second loop. If we don't reinitialize X at this point, then the next time we use it (in Line 18Ø), X will equal 25 instead of the 1 that we need.

Line 17Ø defines a second loop that assigns the numbers 28 down through 5 to the variable Y. Each time, the value of Y will decrease by 1.

Line 18Ø sets up the row counter X for the second loop.

Line 19Ø CALLS the horizontal character display for the second diagonal. This time, on the first pass of the loop, X = 1 and Y = 28. The second pass will assign 2 as the value of X and 27 as the value of Y, and so on, down to X = 24 and Y = 5.

Line 2ØØ is the NEXT statement that accompanies the FOR statement in Line 17Ø. It works much the same as the NEXT statement in Line 15Ø, except that instead of adding 1, it subtracts 1 from the current value of Y.

NESTED LOOPS

It is possible to team several loops in still another way. All of one loop may be part of another loop or group of loops. When we place one loop inside another, we call it a **nested loop.** An example:

```
100 CALL CLEAR
110 FOR A=1 TO 2
120 PRINT "THIS IS LOOP A";A
130 FOR B=1 TO 5
140 PRINT "THIS IS LOOP B";B
150 NEXT B
160 NEXT A
RUN
THIS IS LOOP A 1
THIS IS LOOP B 1
THIS IS LOOP B 2
THIS IS LOOP B 3
THIS IS LOOP B 4
THIS IS LOOP B 5
THIS IS LOOP A 2
THIS IS LOOP B 1
THIS IS LOOP B 2
THIS IS LOOP B 3
THIS IS LOOP B 4
THIS IS LOOP B 5
** DONE **
```

In this example, loop B, which repeated five times, is nested inside loop A, which repeats twice. You will notice that although the FOR statement for loop A comes before the FOR statement for loop B, the order of the NEXT statements is reversed. The NEXT B must come before the NEXT A for the B loop to nest within A loop. See what happens if you change Lines 150 and 160 to read:

```
150 NEXT A
160 NEXT B
```

When you RUN the program, it will display:

```
THIS IS LOOP A 1
THIS IS LOOP B 1
THIS IS LOOP A 2
THIS IS LOOP B 1
* CAN'T DO THAT IN 160
```

When using nested loops, you must be careful not to mix the FOR of one loop with the NEXT of another. The second loop must always be nested completely within the first.

A Second Side of a Triangle

In the last chapter we used a READ/DATA format to construct a triangle composed of asterisks. We can use a nested loop to do the same thing.

```
100 FOR P=1 TO 9
110 FOR H=1 TO P
120 PRINT "*";
130 NEXT H
140 PRINT
150 NEXT P
RUN
```

```
*
**
***
****
*****
******
*******
********
*********
** DONE **
```

Besides nesting inside the P loop, the terminal value of loop variable H is redefined by each pass of the P loop.

Line 1ØØ defines the outer P loop.

Line 11Ø defines the inner H loop. The loop is initialized at 1 and has a terminal value equal to the current value of P. Thus, the first time through the P loop, the H loop is defined as 1 to 1, while the second time, it is defined as 1 to 2, etc.

Line 12Ø PRINTs an asterisk. The semicolon is used to format the display. As H is redefined, the line of asterisks increases by one. Each line of asterisks represents a complete H loop.

Line 13Ø contains the NEXT H statement, nesting it inside the P loop.

Line 14Ø When the inner H loop has finished printing out the lines of asterisks, the computer exits the loop and encounters the second PRINT statement. As there is nothing to PRINT and there is no semicolon, the computer PRINTs nothing. This line gives the next pass of the inner loop, H, a new line on which to PRINT.

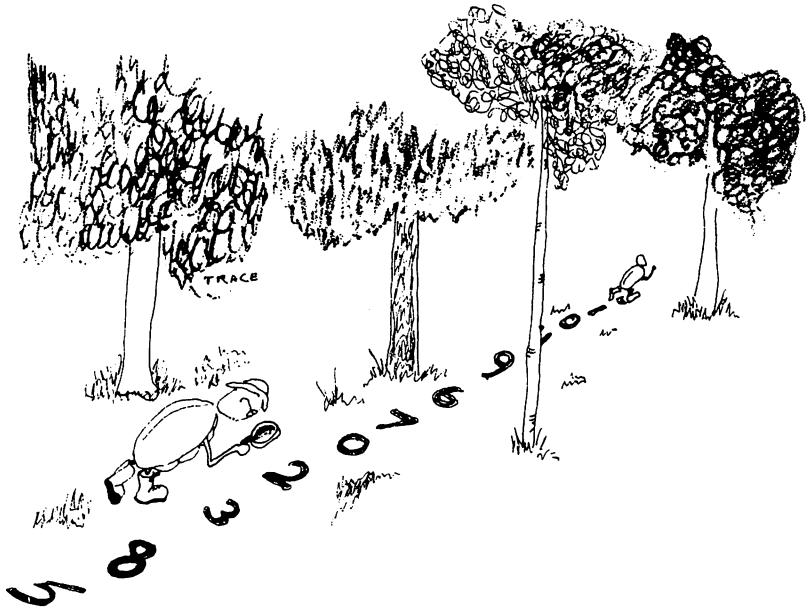
Line 15Ø increments the outer P loop, by 1 and transfers control back to the FOR statement in Line 1ØØ, and the loop begins again.

There is no limit to the number of loops you can nest. You can make this program print three triangles instead of one by nesting both loops inside a third. To do this, add the following lines:

```
9Ø FOR D=1 TO 3

16Ø NEXT D
```

Used alone, FOR-NEXT loops are a powerful programming tool. When combined with branching statements, conditional logic, and other programming statements and functions, FOR-NEXT becomes even more potent.

12 Debugging Programs



Computer programmers call errors that keep their programs from working, **bugs.** Bugs are famous for their ability to turn normally stable programmers into frenzied lunatics.

Program bugs can be hard to find. Sometimes they are simple typing mistakes—the letter "O" where a zero should be, or a lower-case "l" for the number one. Computers are fussier than English teachers about misplaced parentheses or quotation marks; they're real sticklers about incorrect punctuation. Your 99/4A even makes much ado about nothing: forgetting an empty space after a line number or on either side of a BASIC reserved word, you may have noticed, spawns bugs. Because LISTed programs appear with the correct spacing, LISTing can help you figure out where you really need spaces. Check your *User's Reference Guide* for the correct form of any statement or command.

Some commands, such as those used with cassette tape decks, won't work if typed in lower-case letters. In LISTed programs, all statements and reserved words appear in upper-case letters. It is a good programming practice to leave the ALPHA LOCK down and write your programs in upper case, except when enclosing text in quotation marks.

Another programming error results from the use of numeric variables which return values not allowed by the computer. This kind of error occurs most frequently with the "CALL" statements associated with the use of graphics and sound. Each CALL statement has its own parameters which define the range within which all values entered into the statement must fall. If the use of program logic returns values for variables outside of the legal parameters, then a BAD VALUE error message results. In such instances, the error often lies in a program line different from the one specified in the error message. This is because the computer will often accept a range of values when they are first introduced but will balk when it is asked to apply these values to a specific statement in a later portion of a program. The best defense against this kind of error is to become thoroughly familiar with the appropriate range of values allowed in TI BASIC statements and functions. In addition, you need to scrutinize your programming logic to make sure that it does not yield such values.

The computer generally responds to errors in one of two ways. It may immediately reject a new program line when you try to enter it. If this happens, you will have to type your line in again, with corrections. The computer may also interrupt a program you are trying to RUN and display an error message. When this happens you must hunt down the fly in the ointment. Each statement in TI BASIC can fall prey to mistakes peculiar to the way it works.

Perhaps the toughest bug to find is one that doesn't stop the program or print an error message, but simply prevents the computer from doing exactly what you want it to do. In such cases, you must play detective and search through the program for clues to the bug 'whodunit.'

Fortunately, TI BASIC does give you some debugging help.

ERROR MESSAGES

Error messages are your first line of defense against programming bugs. As previously mentioned, error messages occur when you try either to enter a new line or while RUNning a program. Error messages can also occur when the computer is preparing a "symbol table" prior to running a program, before actually executing program steps. The symbol table

is the area in memory where the computer stores functions, arrays, and variables. To do this, it must scan your program, and it can pick out some errors in the process.

Although error messages help some, keep in mind that you may still have to hunt for the source of the error: the problem may not really lie where the warning signal appears. Check pages III-8 to III-12 in your *User's Reference Guide* for a complete list of error messages.

TRACE AND UNTRACE

TRACE and UNTRACE may be used either as commands outside a program or as numbered program statements.

TRACE can help you find errors in program logic by letting you follow each step in a program as it RUNs. Each line number is displayed on the screen within brackets as it is performed. Previously performed steps scroll continuously off the top of the screen while new line numbers appear at the bottom.

TRACE may shift the location on the screen of PRINTed material or graphics displays or it may cover them up with line numbers.

Try TRACEing the program, "Diamond Track" from Chapter 5 to see this command in action. (Load or retype the program, then enter TRACE and RUN.) Because "Diamond Track" has a sort of built-in "trace," you will see line numbers on both the left and right sides of the screen; those produced by TRACE will be bracketed. The TRACE command will also show the execution of Line 14Ø (GOTO 1Ø) not shown by the original program.

When you TRACE a long and complex program the results can be mystifying, as a steady stream of line numbers scroll up the screen. What you really want to see is a specific section that might contain a suspected bug—hard to do when everything on the screen is moving.

RUN, BREAK, and CONtinue can come to the rescue. If you add a BREAK statement after the last line of the program section you wish to investigate, you can stop the action and scrutinize the results. For instance, let's say we only want to trace Lines 2Ø to 6Ø in "Diamond Track." Type in a new line:

```
15 TRACE
```

Next, add

```
65 BREAK
```

Now, when you RUN the program, TRACE only works on Lines 2Ø through 6Ø, BREAKing at Line 65. To go on, type CONtinue; the trace will continue until the next time Line 65 comes up. If you wish to stop the TRACE at Line 65 without breaking out of the program, enter the UNTRACE statement on Line 65.

You may use GOTO to view a short section of the program repeatedly as it is TRACEd: erase Line 65 that you've just added and add

```
65 GOTO 2Ø (or whatever line you wish to TRACE from)
```

TRACE will run continuously, displaying only the lines between 2Ø and 65. With luck, these techniques will help you find your bug. (Keep in mind that you can also start RUNning a program at any point by typing RUN followed by the line number where you wish to start.)

PRINT DEBUG

As we mentioned before, misusing variables or bad variable values often create program bugs. The BREAK and PRINT statements allow us to see what value a variable has at a certain stage of a program.

In the following program

```
1ØØ CALL CLEAR

11Ø FOR X=1 TO 33

12Ø CALL HCHAR(12,X,36)

13Ø NEXT X
```

the computer displays a horizontal row of dollar signs across the screen from left to right, but breaks out with the message

```
BAD VALUE IN 12Ø
```

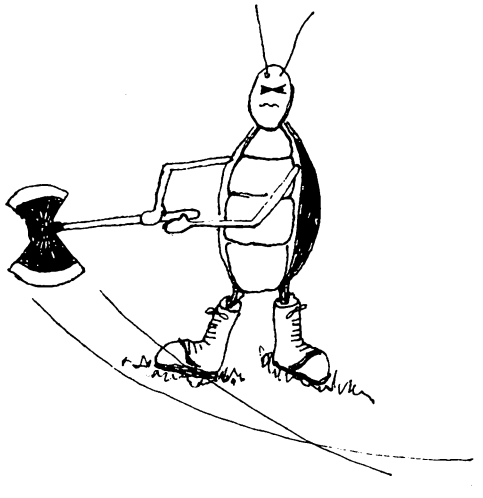
immediately afterward. (This is because the legitimate range for the variable X in this location in the Horizontal CHARacter repetition statement is 1 to 32. The computer has only 32 columns across and will not accept 33 as a good value.) If you type PRINT X after the error message has been displayed, you will see the number "33" displayed—a dead giveaway that something is amiss. Now let's add the following three lines:

```
125 IF X=15 THEN 126 ELSE 126 PRINT X
Ø                         127 BREAK
```

Now our program stops when X has reached a value of 15 and goes to Line 126; the computer PRINTs 15 and the message BREAKPOINT AT 127. If we type CONtinue, the rest of the horizontal line of dollar signs is printed out and, as before, the program breaks out on its own, with a BAD VALUE message. PRINT X now displays the value 33. If a program breaks out with a bad value message, take an educated guess at which variable may be causing the program and ask the computer to PRINT it. If you edit a program line, the computer's memory is cleared of variable values obtained while running a program—so use PRINT before editing lines.

The most difficult bugs to eliminate usually are those which allow a program to run but which prevent the computer from doing what you want. Where variable changes resulting from program action are involved, you can create a "mini-trace" for yourself, investigating the variable in question by simply running that section of the program in a loop and asking that the variable be printed each time.

Delete Lines 125 and 127 in the program above and RUN to witness this effect. The program prints the value of X each time the previous statement using it is executed, moving the display up one line at a time in the process. Notice that the last value entered, 32, is the last legitimate value for X in this situation.



13 Numeric Functions

TI BASIC has a number of built-in numeric functions for complex mathematical operations that you can incorporate into your programs. This chapter examines these functions, particularly the random number and user-defined functions.

INTEGER

INT(X) turns decimal numbers into whole numbers by lopping off the part to the right of the decimal point Thus INT(1.99) equals 1, and INT(1.01) also equals 1. The INT function is often used with the RANDOM NUMBER function.

RANDOMIZE AND THE RANDOM NUMBER FUNCTION (RND)

The BASIC statement RANDOMIZE is used with the RND, or RaNDom number, function to generate a different set of random numbers each time a program is run. Without a RANDOMIZE statement, the RND function will produce the same set of numbers each time the program is run. The next program illustrates this; RUN, then reRUN it and watch the list of numbers. You'll see that the "random" values repeat each time you RUN the program after it BREAKs (Line 140 tells the computer to break whenever C = 1.)

```
100 FOR X=1 TO 7
110 C=INT(9*RND)+1
120 PRINT C
130 IF C=1 THEN 140 ELSE 160
140 BREAK
150 UNBREAK
160 NEXT X
170 PRINT ::
180 GOTO 100
```

If you use CONtinue instead, the computer generates a different series of numbers each time. Reason: RUN starts the program from the beginning, repeating the first few steps; CON continues from the BREAK with subsequent steps that generate different numbers. Try RUNning the program, continue with CON, then reRUN and CONtinue a second time: What happens to the numbers generated after both CONtinue commands?

Another factor which enters into the use of the Random Number function and the RANDOMIZE statement is the use of a **seed.** A seed is established when a value is associated with the RANDOMIZE statements: as, for example, RANDOMIZE 144. Whenever a seed is used, the sequence of numbers generated depends on the number of the seed, and will be the same each time the program is run. Each time the value of the seed is changed, different numbers will be generated. To achieve a truly unpredictable series of numbers, you must use the RANDOMIZE statement, without a seed, with the RND function.

To see what is actually happening with the Random Number function, we can run the following program, and PRINT C to display the actual values generated by the RND function.

```
100 CALL CLEAR

110 RANDOMIZE

120 C=(RND)

130 PRINT C

140 GOTO 110
```

As you can see, the computer produces a series of decimal numbers between 1 and 0. Since such numbers are unwieldy, we usually use RND with the INTeger function to get whole numbers. But if we simply rewrite Line 120 as

```
120 C=INT(RND)
```

we get a set of zeros. This is because RND only generates decimal numbers *between* the integers 0 and 1, and the INTeger function always rounds to the next *smaller* whole number. We can add 1:

```
120 C=INT(RND)+1
```

but this just gives a battery of ones because, after all, we are just adding a one to all those zeros. You must specify a value X within the parentheses with the RND function, to generate a set of random numbers from one to X: NUMBER = INT(X*RND) + 1, where X is the highest value you wish to include in your range of numbers. To set random values between two numbers higher or lower than zero, use INT((Y − X + 1)*RND) + X. For example,

NUMBER = INT((10 − −10 + 1)*RND) + −10.

gives random integers between 10 and −10 inclusive, where X is the lower number, −10, and Y is the higher number, 10.

To illustrate the use of the RND function, we now turn to a program designed to simulate the tossing of a coin. In this case, the computer simply generates integer values of either 1 or 2 at Line 130, C = INT(2*RND) + 1. If C equals 1, then the message "HEADS" is printed. Otherwise, the program goes to Line 180 and prints "TAILS." RUN the program; press any key when the computer asks, "HEADS OR TAILS?"

```
100 CALL CLEAR
110 PRINT "     HEADS OR TAI
LS?" ::
120 CALL KEY(3,R,S)
130 IF S=0 THEN 120
140 RANDOMIZE
150 C=INT(2*RND)+1
160 IF C=1 THEN 170 ELSE 190
170 PRINT ,"HEADS"::
180 GOTO 110
190 PRINT ,"TAILS"::
200 GOTO 110
```

For another program incorporating RANDOMIZE and RND, take a look at "The Random Character Generator" in Chapter 15.

OTHER NUMERIC FUNCTIONS

Absolute Value—ABS(X)

This function gives you the absolute value of a numeric expression, regardless of sign. If the value for X is equal to or greater than zero, ABS(X) equals X. If X is less than zero (negative), ABS(X) is equal to a negative X.

Arctangent—ATN(X)

The arctangent function gives you the arctangent of the value of X as an angle, in radians, whose tangent is X. To convert to the same angle in degrees, multiply the results of ATN(X) by 57.2957795.

Cosine—COX(X)

COS(X) gives you the cosine of the value X as an angle, in radians. To convert an angle in degrees, multiply by 57.2957795. The value of

X must have an absolute value less than 1.5707963266375 times 10^{10} or the program stops and gives you an error message.

Sine—SIN(X)

SIN(X) yields the sine of the value of X as an angle in radians. To convert to an angle in degrees, multiply SIN(X) by 57.2955795. To convert back into radians from degrees, multiply the angle by the reciprocal of this number. If a SIN(X) has an absolute value greater than 1.5707963266375 times 10^{10}, the program stops and prints an error message.

Tangent—TAN(X)

TAN(X) yields the tangent of the value of X as an angle, in radians. To convert to an angle in degrees, multiply by 57.2957795. Again, if TAN(X) has an absolute value greater than 1.5707963266375 times 10^{10}, you get an error message.

Exponential—EXP(X)

EXP(X) delivers the value of X^e where e = 2.718281828. The EXPonential function is the inverse of the Natural Logarithm, or LOG, function. Hence X = LOG(EXP(X)).

Natural Logarithm—LOG(X)

LOG(X) delivers the natural logarithm of the value X. The LOG function is the inverse of the exponential, or EXP, function. Hence, X = EXP(LOG(X)). If the value of X is equal to or less than zero, the program stops and gives you an error message.

Square Root—SQR(X)

SQR(X) delivers the square root of the value X. If X is less than zero, an error message results.

Signum—SGN(X)

SGN(X) delivers the algebraic sign of the value X. If X = 0, then, SGN(X) is zero. If X is greater than zero, then SGN(X) = 1. If X is less than zero, then SGN(X) = −1.

USER-DEFINED FUNCTIONS

If you cannot find a built-in function for your needs, and you are

writing a program that performs one or more specific operations many times, you may wish to create your own functions with the DEFine statement.

To use DEFine, simply type "DEF" followed by a function name. Follow the same rules in naming functions that you'd use for variables. You may follow the function name by a parameter enclosed in parentheses; the parameter must also be a valid variable name. Then make your newly defined function equal to a numeric expression. For example, if we wished to create a function to calculate the cubes of numbers, we might write:

```
100 DEF CUBE(X)=Y*Y*Y
```

where CUBE is our function name, (X) is our parameter, and the results are equal to Y*Y*Y. The following program prints the cubes of numbers from 1 up.

```
100 DEF CUBE(X)=Y*Y*Y

110 Y=1

120 PRINT CUBE(X)

130 Y=Y+1

140 GOTO 120
```

If we add:

```
115 PRINT Y
```

and change Line 140 to GOTO 115 (instead of 120), the computer PRINTs each new value of Y following by the corresponding value for CUBE(X).

Once you have defined a function in your program, you can use it anywhere just by entering its name. If you have not specified a parameter, the function is evaluated with the values of the variables in the statement where it was originally defined. If you do specify a parameter, the result of the numeric expression in which the function appears yields the value of the parameter. The function is then evaluated using the newly derived value of the parameter and the current program values of the variables included in the definition of the function.

The parameter of a user-defined function may have the same name as other variables in a program: using the function does not alter the values of those variables. In other words, the parameter is "local" to

a given function.

The computer doesn't actually perform a function until you refer to it in a numeric expression; it takes no action upon coming across the statement where you DEFine your function. Be sure that you DEFine your function before using it: your definition must have a lower line number than other lines referring to the function.

A few other rules: do not use the name of a function as part of its own definition. For instance, the computer will reject DEF B = B+2 but will accept DEF B = 0+2. If you include a parameter, you must include it every time you refer to your function. If, for instance, we drop the (X) from Line 120 in our CUBE program, the program will not run. Conversely, if you do not use a parameter in the original definition, don't try to add one later. The following program is functionally identical to our original CUBE program, but has no parameter specified.

```
100 DEF CUBE=Y*Y*Y    130 PRINT Y;
110 Y=0               140 PRINT CUBE
120 Y=Y+1             150 GOTO 120
```

If we were to add (X) to CUBE on Line 140, the error message, NAME CONFLICT IN 140, would result.

You can DEFine string functions as well as numeric ones. If so, follow the rules for manipulating strings and string variables.

Now let's put together some of these functions in a short demonstration program with INTeger, RND (random number), ABS (absolute value), and SGN (signum). The program produces a series of random numbers from 19 to −20 on the left hand side of the screen and plays various tones, depending on the values of the numbers displayed.

```
100 CALL CLEAR                    170 GOTO 120
110 RANDOMIZE                     180 CALL SOUND(200,1600,0)
120 X=INT(40*RND)-20              190 GOTO 120
130 PRINT X                       200 FOR Y=1 TO 4
140 IF ABS(X)>16 THEN 200         210 CALL SOUND(100,800,0)
150 IF SGN(X)=-1 THEN 160 EL      220 NEXT Y
SE 180                            230 GOTO 120
160 CALL SOUND(200,120,0)
```

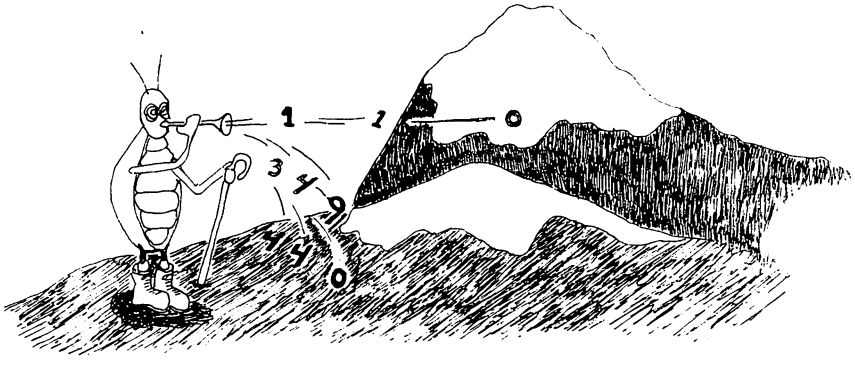
Lines 11Ø-12Ø Together, the RANDOMIZE statement and the RND function ensure a random sequence of numbers. The INTeger function in Line 12Ø gives us whole numbers. We never get 20 though, because the INT function always rounds down.

Line 14Ø If the absolute value of any number printed is greater than 16, then the program skips to Line 2ØØ and produces four quick tones.

Line 15Ø If any of the numbers not caught and rerouted by Line 14Ø have a −1 for a sign from the SIGNUM function, then the computer makes the low tone requested in Line 16Ø. Numbers with a positive sign will cause the high-pitched sound defined in Line 18Ø.

Lines 16Ø, 18Ø, and 21Ø The CALL SOUND statements in these lines produce tones determined by the values of the various numeric functions.

14 Computer Sound and Music



THE SOUND CHIP

The TI-99/4A comes equipped with a sound chip capable of generating music with three-part harmony and a wide variety of sound effects. The designers of the chip have gone to a lot of trouble to build in labor-saving features which facilitate sound programming. The most advantageous feature of all is that the chip operates independently, meaning that while a sound is playing, the computer can go off to another part of a program and perform other tasks simultaneously. This feature is particularly useful when combining sound and graphics.

There is only one sound command in TI BASIC: CALL SOUND. CALL SOUND is actually a sound sub-program that instructs the computer to produce tones with a frequency, volume, and duration that you choose. A tone's frequency is measured in number of vibrations, or cycles, per second, also called **Hertz** (Hz). The TI has a frequency range from 110 Hz at the low end (the A two octaves below the middle C on a piano) to a high of 44,735 Hz, well above the range of human hearing which ranges from about 20 Hz to about 20,000 Hz. The human voice has a range from about 125 to 1,000 cycles per second, while a concert grand piano has a range of 27.5 to 4,186.

The following sound program demonstrates the TI's frequency range

by producing tones from 110 to 25,000 cycles per second. You can use this program to test your own hearing range: RUN the program, and when you no longer hear the tone, press FCTN:CLEAR to stop the program. Then type PRINT FREQ and press ENTER. The TI will display the frequency of the note it was playing when you stopped the program. (Warning: this is *not* a true hearing test, as the results will be affected by your TV or monitor speaker.)

```
100 FOR FREQ=110 TO 25000 ST
EP 20
110 CALL SOUND(-110,FREQ,3)
120 NEXT FREQ
```

The TI has 30 volume settings: 0 is the loudest and 30 is the softest. Here is a program demonstrating these settings:

```
100 FOR VOL=0 TO 30
110 CALL SOUND(400,440,VOL)
120 NEXT VOL
```

The period of time that a tone lasts is called its duration and is controlled by a duration setting in milliseconds. (One thousand milliseconds are equal to one second.) A single CALL SOUND statement may be programmed for a duration between one and 4,250 milliseconds.

This sample program plays six tones, each one lasting twice as long as the one before.

```
100 READ DUR
110 CALL SOUND(DUR,110,3)
120 IF DUR=3200 THEN 170
130 FOR SILENCE=1 TO 600
140 NEXT SILENCE
150 GOTO 100
160 DATA 100,200,400,800,160
0,3200
170 END
```

CALLING SOUND

Programming the values to be entered into a CALL SOUND statement is a simple operation. The proper format for programming a CALL SOUND statement using one, two and three voices appears below:

One Voice:	CALL SOUND(duration,frequency 1,volume 1)
Two Voices:	CALL SOUND(duration,frequency 1,volume 1, frequency 2,volume 2)
Three Voices:	CALL SOUND(duration,frequency 1,volume 1, frequency 2,volume 2,frequency 3,volume 3)

The duration value comes first and controls the duration of all the tones in the statement. To program duration, just enter a number between 1 and 4,250; the note will last for that number of milliseconds.

Then you enter a frequency for Voice 1, followed by its volume setting. Remember: the computer's frequency values range from 110 through 44,733 Hz. You'll find a frequency table for four full octaves of musical tones on page III-7 of your *User's Reference Guide.* Feel free to make up your own tuning systems; this is one of the major strengths of your 99/4A.

Setting the volume for Voice 1 is next. All you do is select a volume setting from 0 to 30. This setting can be further controlled through the use of the volume dial on your TV or monitor. Remember that 0 is the loudest setting and 30 is the softest.

Programming the frequency and volume for the second and third voices is just as easy. Simply add the new frequencies and volumes to the CALL SOUND statement. A maximum of three musical voices may be obtained.

NOISE SETTINGS

Until now we have been discussing the production of pitched tones: those that vibrate regularly at a certain number of cycles per second. A sound that vibrates erratically or randomly is called noise. The sound chip in your TI has eight pre-programmed noises for creating special sound effects. Here is a program that plays all eight noises.

```
100 FOR EFFECT=-8 TO -1
110 CALL SOUND(200,EFFECT,4)
120 NEXT EFFECT
```

These effects are programmed by entering a negative frequency value from −1 to −8 inclusive. Settings of −1 through −4 give **Periodic Noise** effects, while settings of −5 through −8 yield **White Noise.**

Duration and volume settings for sound effects are controlled the same way as musical tones. You may even add a sound effect to a CALL SOUND statement containing musical tones:

```
CALL SOUND(1000,440,3,349,3,
330,3,-8,4)
```

This statement will produce three musical tones and one noise all played together for one second.

NEGATIVE DURATION VALUES

When you CALL SOUND, the computer follows a special protocol for running programs. Because the sound chip operates independently, the computer can perform other program statements while a CALL SOUND statement is being executed. If you've used more than one CALL SOUND statement, the computer finishes the first before going on to the next. Some sound effects, however, operate more smoothly, without annoying clicking, when one statement can flow into another. In other words, you'll want to interrupt one CALL SOUND by another before the first statement is completed. You do this with a negative duration setting, which instructs the computer to cut off the previous sound statement and execute the new one.

Try these two samples. The first uses a normal duration setting, and the second, a negative duration setting; listen to the difference.

```
100 FREQ=220
110 FOR SAMPLE=1 TO 15
120 CALL SOUND(100,FREQ,3)
130 FREQ=FREQ+20
140 NEXT SAMPLE
```

```
100 FREQ=220
110 FOR SAMPLE=1 TO 15
120 CALL SOUND(-100,FREQ,3)
130 FREQ=FREQ+20
140 NEXT SAMPLE
```

You may have noticed that the second program ran much faster than the first. In the second program, the computer did not wait for each sound statement to finish. It simply cut off the sound which was playing as soon as it encountered the negative duration in the following sound statement.

PROGRAMMING A SONG FOR ONE VOICE

Now that you understand the CALL SOUND settings, let's move head and write a program for a song using one of the TI's three musical voices. Whenever you are doing sound programming, you should consider the context in which the sound is used. If it is strictly a sound program, then one programming strategy might be to enclose the CALL SOUND statement in a READ/DATA program format. If, however, you are using sound with graphics, a series of CALL SOUND statements interspersed with graphics statements would prove more flexible. There is no one way of programming sound—several different programming strategies may yield very similar results. Here is a program that uses two different programming strategies to play the same phrase.

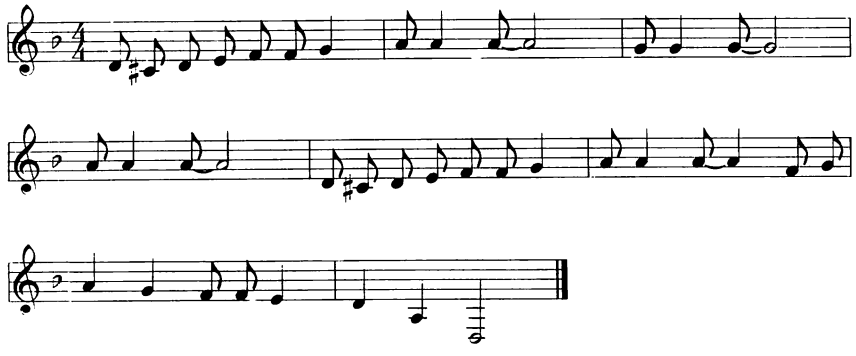
```
100 REM ***THIS SECTION USES
CALL SOUND IN SERIES***
110 CALL SOUND(150,131,3)
120 CALL SOUND(150,165,3)
130 CALL SOUND(150,196,3)
140 CALL SOUND(150,262,3)
150 CALL SOUND(150,196,3)
160 CALL SOUND(150,165,3)
170 CALL SOUND(150,165,3)
180 REM ***THIS SECTION USES
READ/DATA ***
190 READ DUR, FREQ
200 CALL SOUND(DUR,FREQ,3)
210 IF DUR=600 THEN 230
220 GOTO 190
230 END
240 DATA 150,131,150,165,150
,196,150,262,150,196,150,165
,600,131
```

For straight music programming, the READ/DATA format is a lot easier because you don't have to keep typing CALL SOUND.

Let's use a modified version of the READ/DATA format to program a song, "Joshua Fought the Battle of Jericho," a traditional spiritual. The explanation that follows assumes that you can read music. If not, skip ahead to "Programming Sound Effects," which requires no music theory.

JOSHUA FOUGHT THE BATTLE OF JERICHO

(Traditional)



```
100 READ DUR,FREQ
110 IF DUR=0 THEN 180
120 CALL SOUND(DUR*150,FREQ,3)
130 GOTO 100
140 DATA 1,294,1,277,1,294,1
,330,1,349,1,349,2,392
150 DATA 1,440,2,440,5,440,1
,392,2,392,5,392,1,440,2,440
,5,440
160 DATA 1,294,1,277,1,294,1
,330,1,349,1,349,2,392
170 DATA 1,440,2,440,3,440,1
,349,1,392,2,440,2,392,1,349
,1,349,2,330,2,294,2,220,4,1
,47,0,0
180 END
```

The general idea behind this method of programming music: the duration and frequency of each note in the song are treated as separate variables. We have chosen 150 milliseconds as the duration of an eighth note. By setting up a CALL SOUND statement that reads the duration as DUR*150, we can vary each note's duration by changing the value of DUR. Since time is proportional, this makes it easy to figure out the duration values for each note. The table below illustrates the musical notation equivalents for the duration values used here.

1*DUR = eighth note

2*DUR = quarter note

3*DUR = dotted quarter note

4*DUR = half note

By placing the duration number and the frequency number (from the Frequency Table) into a DATA bank, we can play the notes in the proper order with the correct frequency and duration values.

With this in mind, see if you can figure out how the program works.

Line 1ØØ	READs the first two pieces of DATA from the DATA list and assigns them to the variables DUR and FREQ. The first item in DATA is the duration value and the second is the frequency.
Line 11Ø	checks the duration value to see IF it is Ø. IF it is Ø, THEN the computer skips to Line 18Ø. IF it is not Ø, THEN the computer moves on to Line 12Ø.
Line 12Ø	CALLs the SOUND: the value of DUR is multiplied by 150, the total number of milliseconds of duration; the value of FREQ is taken as the frequency number; the volume remains constant at 3.
Line 13Ø	transfers control back to the READ statement in Line 1ØØ. This creates a loop for READing and playing the notes.
Lines 14Ø-17Ø	contain the DATA for the duration and frequency of each note. The last two pieces of information (Ø,Ø) are flags to indicate the end of the DATA list.
Line 18Ø	ENDs the program.

RESTORE FOR REPEATS

The RESTORE statement can be used along with counters and DATA flags to repeat sections of music without your having to go through the trouble of typing them in twice. If you add the lines below to your program, the entire piece will play twice.

First, change Line 18Ø so that it reads:

```
18Ø FLAG=FLAG+1
```

Then add these lines:

```
19Ø IF FLAG=2 THEN 22Ø        21Ø GOTO 1ØØ

2ØØ RESTORE                   22Ø END
```

The FLAG counter in Line 18Ø keeps track of how many times the DATA has been used. Line 19Ø checks the value of the FLAG. IF it has been used once, THEN DATA will be restored and the whole piece will play over. IF it has been used twice, THEN the piece will END.

A SONG FOR THREE VOICES

Using all three musical voices makes the music sound much stronger. You can approach programming for three voices in a number of ways. The next program uses the method we have found to be flexible and easy to use, as well as giving us accurate control of individual notes.

The program uses three CALL SOUND statements, although only one is used at any given moment during the music. The first CALL SOUND is used when there is note information for three voices. The second is used when there is note information for two voices, and the third is used when only one voice is required. The program incorporates a READ/DATA format for processing frequency and duration values.

In this program, we used 150 milliseconds for a sixteenth note: thus, a 1 in the DATA statement represents a sixteenth note. Similarly, a 2 represents an eighth note and a four a quarter note. A zero is used as a frequency value when the voice is not used for tone production.

Type in this rather long program, and you will will be rewarded with the music of the 99/4A's sound chip at its best. A line-by-line description follows the program. If you have any trouble understanding how the program works, we suggest you study this commentary.

```
100 CALL CLEAR
110 PRINT TAB(9);"LILI MARLE
NE" ::
120 PRINT "CREDITED TO NORBI
T SCHULTZE" ::
130 PRINT "PROGRAM DESIGN AN
D MUSICAL" ::
140 PRINT "ARRANGEMENT BY" ::
150 PRINT "JAMES VOGEL"
160 READ D,F1,F2,F3
170 IF D=0 THEN 400
180 IF (F2=0)*(F3=0)THEN 220
190 IF F3=0 THEN 240
200 CALL SOUND(D*150,F1,3,F2
,3,F3,3)
210 GOTO 160
220 CALL SOUND(D*150,F1,3)
230 GOTO 160
```

```
240 CALL SOUND(D*150,F1,3,F2
,3)
250 GOTO 160
260 DATA 3,330,196,131,1,330
,131,0,3,330,247,131,1,349,1
,31,0
270 DATA 4,392,294,233,4,330
,262,0,3,349,220,147,1,349,1
,47,0
280 DATA 3,349,262,208,1,523
,0,0,8,494,349,196,3,294,247
,220
290 DATA 1,294,196,0,3,294,1
96,175,1,330,147,0,4,349,247
,196
300 DATA 3,349,247,147,1,392
,147,0,3,494,294,196,1,440,3
,49,262
310 DATA 3,392,330,110,1,349
,294,220,7,330,196,131,1,262
,131,0
320 DATA 4,440,262,175,3,494
,392,220,1,523,440,0,4,494,2
62,185
330 DATA 4,440,370,220,4,440
,330,196,4,392,330,131,7,494
,330,131
340 DATA 1,440,262,330,4,392
,330,247,4,349,294,196,7,440
,349,247
350 DATA 1,392,330,247,4,349
,294,208,4,330,262,247,7,392
,262,196
360 DATA 1,330,262,123,7,392
,330,110,1,349,294,247,4,349
,294,196
370 DATA 4,587,494,349,8,523
,392,165,4,392,262,165,4,330
,262,110
380 DATA 7,392,262,220,1,349
,208,0,4,349,247,220,4,247,1
96,175
390 DATA 12,262,165,196,0,0,
0,0
400 END
```

Here is the line-by-line description.

Lines 1ØØ-15Ø	format the print display. This is optional.
Line 16Ø	READs the first four pieces of note information: duration, Voice 1 frequency, Voice 2 frequency, and Voice 3 frequency, respectively. These values are assigned to the variables DUR, F1, F2, and F3 respectively.
Line 17Ø	checks the current duration value. IF it is Ø, control transfers to Line 4ØØ. IF it is not Ø, THEN control passes to Line 18Ø.
Line 18Ø	checks the current frequency values of Voices 2 and 3. IF both values are zero (meaning that only Voice 1 is being used), THEN control transfers to Line 22Ø (the CALL SOUND for one voice). If the values are not both 0, THEN control passes to Line 19Ø.
Line 19Ø	checks the current frequency value for Voice 3. IF it is Ø (meaning Voice 3 is not used), THEN control transfers to Line 24Ø (the CALL SOUND for two voices). IF it is not 0, THEN control passes to Line 2ØØ.
Line 2ØØ	contains the CALL SOUND statement for three voices.
Line 21Ø	loops back to Line 16Ø to READ the next four pieces of DATA.
Line 22Ø	contains the CALL SOUND statement for one voice.
Line 23Ø	loops back to Line 16Ø to READ the next four pieces of DATA.
Line 24Ø	contains the CALL SOUND statement for two voices.
Line 25Ø	loops back to Line 16Ø to READ the next four pieces of DATA.

Lines 16Ø-25Ø make up the backbone of the program. They process the note information and determine how many voices will play.

Lines 26Ø-39Ø	form the DATA bank containing the duration value and the frequency values for groups of three notes. Since the READ statement will always read four values, values must be entered even if the voice is not used; hence a Ø for the frequency of an unused voice. A duration of Ø represents a flag to leave the READ loop (see Line 17Ø).
Line 4ØØ	ENDs the program.

If you want to program a moment of silence, we suggest that you actually program a very high frequency, such as 44,000. This will allow you to remain in the READ/DATA format and continue processing note

information. Although the tone will be played, it is way beyond the range of human hearing and will therefore pass for silence.

This program demonstrates only one of the many ways you can program the sound chip. Try composing a song of your own.

SOUND EFFECTS

Programming the sound chip for effects is easy, and can produce some interesting results. Here are a few which you can use as models.

```
100 REM SONAR
110 FOR A=1 TO 3
120 FOR VOL=0 TO 25
130 CALL SOUND(-25,-1,VOL)
140 FOR DELAY=1 TO 20
150 NEXT DELAY
160 NEXT VOL
170 NEXT A
```

This program uses nested loops to help create the effect. The A loop makes the effect occur three times. The VOL loop enters a series of 25 volume settings which gives the effect its fade away quality. The DELAY loop spaces out each sonar sound, giving the echo effect.

We used a -1 frequency setting for the pre-programming tone and a -25 duration setting to make the sound statements blend together.

Here is a similar effect which uses the same program format. We used a -5 freuqency setting and a -250 duration for dramatically different results.

```
100 REM DOOM
110 FOR A=1 TO 3
120 FOR VOL=0 TO 25
130 CALL SOUND(-250,-5,VOL)
140 FOR DELAY=1 TO 25
150 NEXT DELAY
160 NEXT VOL
170 NEXT A
```

The next program demonstrates a siren. It is created with the help of the STEP command and two FOR-NEXT loops. To exit this program, press the FCTN:CLEAR, as the program runs in a continuous loop.

```
100 FOR X=800 TO 1100 STEP 1
0
110 CALL SOUND(-150,X,0)
120 NEXT X
130 FOR Z=1100 TO 800 STEP -
10
140 CALL SOUND(-100,Z,0)
150 NEXT Z
160 GOTO 100
```

A combination of a steady rise in frequency coupled with a negative duration value, which blends the tones together, produces the rising pitch. The STEP statement causes the descending pitch.

Using one loop to control frequency and a second for volume can create interesting arcade effects.

```
100 FOR T=2000 TO 800 STEP-2
5
110 CALL SOUND(-150,T,0)
120 NEXT T
130 FOR Z=0 TO 30
140 CALL SOUND(250,-7,Z)
150 NEXT Z
```

We suggest that you experiment with different frequency, volume, duration, and STEP values to create other effects using similar programming formats. You will be surprised how dramatically an effect can change when you alter a single number.

SOUNDING A FINAL NOTE

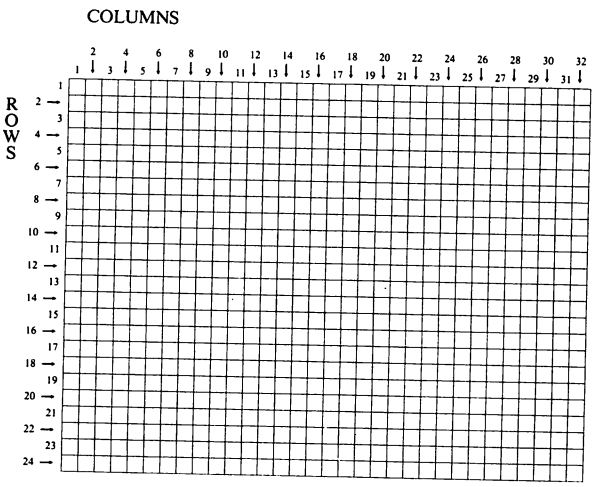
If you are interested in further exploring the possibilities of the TI sound chip, go on to Chapter 20. There we show you how to turn your computer keyboard into a touch-sensitive organ keyboard. And you'll find out how to mix sound with graphics.

15 BASIC Graphics

In this chapter we will explore the TI BASIC set of statements that put images and colors at your fingertips. With practice, you'll soon be creating your own spectacular displays.

THE SCREEN

In graphics mode the 99/4A's screen is divided into 32 vertical columns and 24 horizontal rows (Figure 15.1). Each square is called a **character space,** and there are 32 times 24, or 768, of these character spaces on the screen. Because some television sets don't show the extreme left or right columns, when you are typing in programs, you have access to only 28 columns. But when you RUN a graphics program, all 32 columns can be displayed.



Each character space is further broken down into a checkerboard of eight by eight (64) tiny squares called **pixels;** the screen holds altogether 49,152 individual pixels (32 times 24 times 64). All the characters sent out from the keyboard are made up of these pixels. TI BASIC allows you to define your own characters by generating a code that determines which pixels in a character space are "on" or "off."

BASIC GRAPHICS STATEMENTS

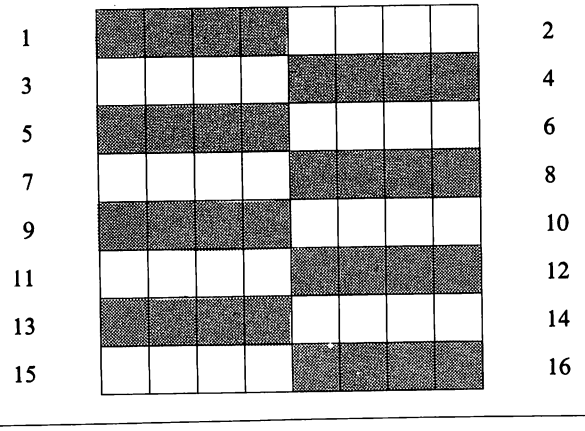
Each graphics statement has several **terms** following it in parentheses; these terms let you specify exactly what you want the statement to do and where. The correct form for each statement is given below.

CALL CHAR (character code, pattern identifier)

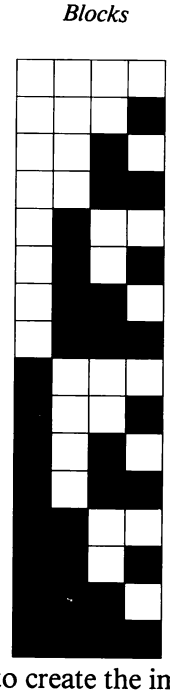
The CALL CHAR statement defines a character space by means of the two terms in parentheses: a character name, or a **character code;** and a **pattern identifier.** The appendix of your *User's Reference Guide* lists the character codes for all the keyboard symbols, which are assigned numbers from 32 to 127. In addition to these codes, 32 blank character codes from 128 to 159 are available for you to define yourself.

You can even redefine the normal character spaces of the keyboard symbols by using their character codes in a CALL CHAR statement. For instance, you can make the number "6" represent a character you choose by using its character code (54) with a pattern identifier you create. Any program in which you do this will show a new character in place of "6" whenever you use character code 54 in a program line.

The 64 pixels in each character space are functionally divided into 16 horizontal sections of four pixels each that are read like an ordinary book page from left to right and top to bottom:

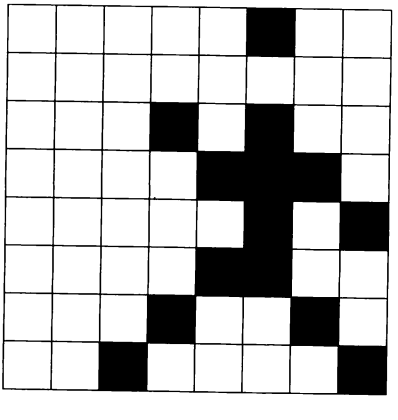


To define the pixels in a character, you must use **block codes.** Each of the four-pixel horizontal bars in a character space can have 16 different combinations of light and dark squares. Hence there are 16 different block codes, one for each possibility.



Let's say we wanted to create the image of a running figure and have it appear in a program instead of the keyboard symbol "6." We could write

```
CALL CHAR(54,"0400140E050C12
21")
```



You must be certain to enclose the pattern identifier in quotes when writing CALL CHAR and to use only upper-case letters for the ABCDEF of the block codes. To see the new character we have created, we can enter the following simple program:

```
100 CALL CHAR(54,"0400140E05
0C1221")
110 CALL HCHAR(12,16,54)
300 GOTO 110
```

After typing RUN, the image of the running man appears at row 12 and column 16 on the screen. Notice that the "6" in the column number after the HCHAR statement in Line 110 is also a running man because we have redefined the character space for the number 6 for the duration of the program. If you press FCTN:CLEAR, both images go back to "6."

CALL CLEAR

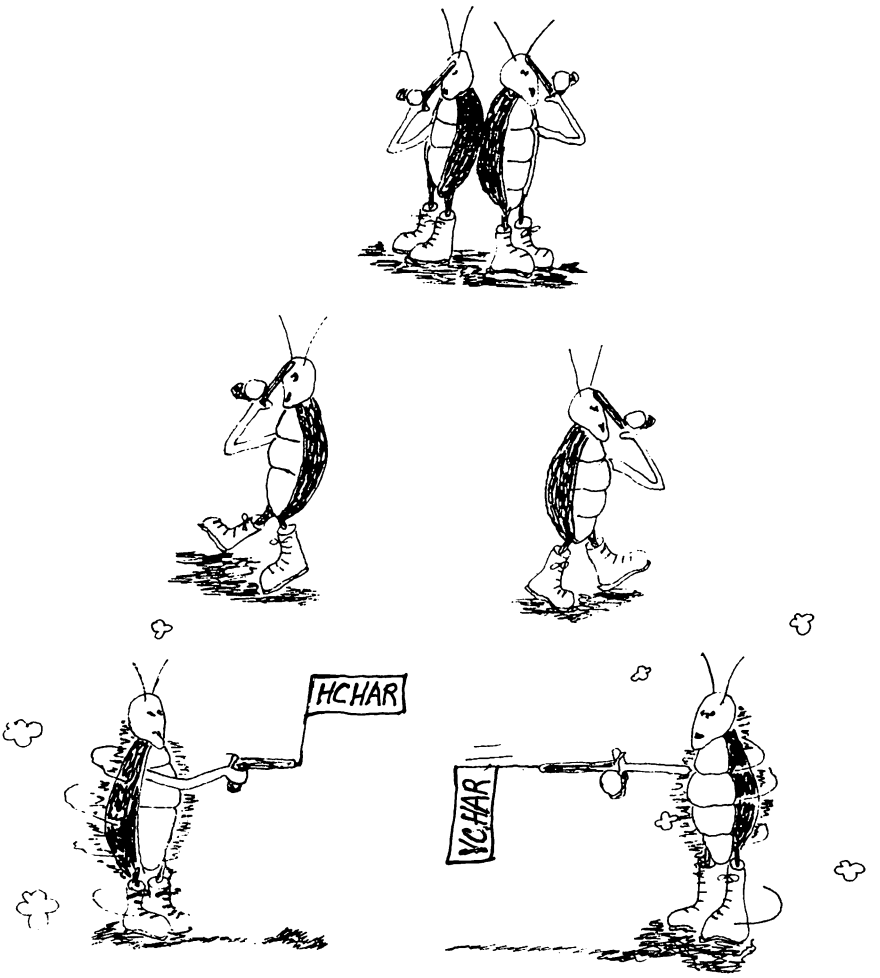
Whenever you use CALL CLEAR, the computer places a blank in all the character spaces on the screen, erasing whatever was there before. For this reason, CALL CLEAR is often a program's first statement. It is useful whenever you want to switch from one set of images to another. It can also help create the illusion of motion—something we'll discuss at greater length later.

Try entering

```
90 CALL CLEAR
```

in our little CALL CHAR program. You can go one step further and redefine the blank space (character code 32) with still another character; if you do, you will fill the screen with that character when you use CALL CLEAR. Try it by changing the 54 in Line 100 to a 32. Can you tell why the 6 is still in the center of the screen?

CALL HCHAR and CALL VCHAR
(row, column, character code, number of repetitions)



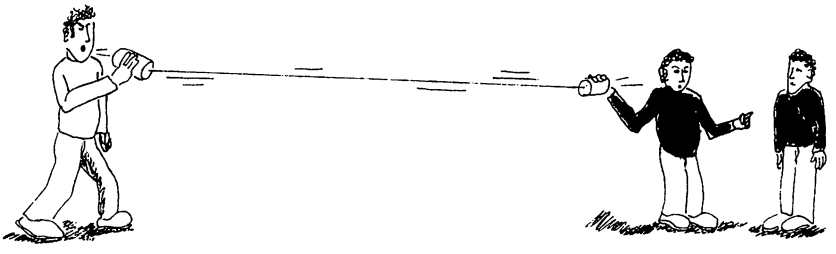
CALL HCHAR and CALL VCHAR display characters on the screen. These two statements work much the same way, except that CALL HCHAR displays character repetitions horizontally, and CALL VCHAR displays them vertically. The first two terms in these commands specify at which row and column number you want your character to start. The third term specifies the character code—either one that displays a keyboard symbol or a new one you have defined in a CALL CHAR statement.

You must limit yourself to the number of screen rows and columns when filling in the first two terms of a CALL HCHAR or VCHAR statement. The values for character codes, however, may run from 0 to 32767 inclusive. But as a rule, you'll only use the values of the normal ASCII character code list.

The last term of the HCHAR and VCHAR statement represents the number of times you wish your character repeated. Again, the computer can accept up to 32767 repetitions, but since 768 repetitions fill the entire screen, you'll see no change after the screen has filled once.

Go back to Line 11Ø, add the number 2ØØ after the character code 54, and reRUN the program. The computer displays 2ØØ images of a "6" horizontally, starting at row 12 and column 16. Now substitute a "V" for the "H" in Line 11Ø.

CALL COLOR (character set, foreground color, background color)



The CALL COLOR statement allows you to change character foreground and background color on the screen. The dots that actually form a given character are in the foreground color; the rest of the character space is in the background color. A third layer underlying both foreground and background is the screen color. The values for the 16 colors your computer can generate are:

Value	Color	Value	Color
1	Transparent	9	Medium Red
2	Black	10	Light Red
3	Medium Green	11	Dark Yellow
4	Light Green	12	Light Yellow
5	Dark Blue	13	Dark Green
6	Light Blue	14	Magenta
7	Dark Red	15	Gray
8	Cyan	16	White

If you use transparent (1) as either a foreground or background color, then whatever color the screen is will show through your character. If you specify transparent for both the foreground and background color, your character will be invisible. If you don't CALL a COLOR, the foreground will be black and the background will be transparent.

CALL COLOR statements affect the color of **character sets** containing eight characters. You must specify the correct character set to change a single character's color; conversely, all characters in that set change color with one CALL COLOR statement. The computer uses 16 different character sets, each corresponding to a section of the character code list:

Set	Character Codes	Set	Character Codes
1	32-39	9	96-103
2	40-47	10	104-111
3	48-55	11	112-119
4	56-63	12	120-127
5	64-71	13	128-135
6	72-79	14	136-143
7	80-87	15	144-151
8	88-95	16	152-159

Let's say you wanted to use an HCHAR or VCHAR statement to display a white 6 on a black background. You'll recall from the character code list that the code for 6 is 54, which, from the table, falls into character set 3. So you would write:

```
100 CALL CLEAR
110 CALL COLOR(3,16,2)
120 CALL VCHAR(12,16,54,200)
130 GOTO 110
```

where 16 (white) is the foreground color, 2 (black) is the background, and 3 is the character set containing the number 6.

Remember: each CALL COLOR statement only affects the eight characters within a single character set; moreover, only one set may be specified per statement. To change the color of characters from different character sets, you must write separate CALL COLOR statements for each set. If you specify a value of one for your character set, all of the empty spaces on the screen will change to the background color because the empty space is included in character set one.

CALL SCREEN (color-code)

You probably noticed that the short CALL COLOR program in the previous section produced white characters on a black background, but left the screen surrounding the characters green. The CALL SCREEN statement changes screen color so you can match the screen to the character background. The underlying screen color shows through in any space not affected by an HCHAR or VCHAR command. It also shows through if the foreground or background color is transparent.

To use CALL SCREEN merely put in parentheses one of the same color code values you used in the CALL COLOR statement. For a solid black background in our last program we would insert

```
105 CALL SCREEN(2)
```

The following program puts the CALL COLOR and CALL SCREEN statements to work. Numbers from character set 3 are displayed on the screen by four CALL COLOR statements at Lines 100, 130, 160, and 190. A subroutine in Lines 230-320 prints the numbers in each new set of colors. Line 220 loops the program back to the beginning. Notice that each CALL SCREEN statement features the same color as the background color in the previous CALL COLOR statement, maintaining a solid background over the entire screen.

```
100 CALL COLOR(3,16,13)
110 CALL SCREEN(13)
120 GOSUB 230
130 CALL COLOR(3,13,16)
140 CALL SCREEN(16)
150 GOSUB 230
160 CALL COLOR(3,6,2)
170 CALL SCREEN(2)
180 GOSUB 230
190 CALL COLOR(3,2,12)
200 CALL SCREEN(12)
210 GOSUB 230
220 GOTO 100
230 CALL CLEAR
240 A=7654321Ø
250 B=1234567
260 C=7654321Ø
270 FOR X=1 TO 4
280 PRINT A;B;C;;;;;
290 NEXT X
300 FOR K=1 TO 1000
310 NEXT K
320 RETURN
```

Here is a program outline for the CALL COLOR demo program.

Lines 100-120, 130-150, 160-180, and 190-210 Each one of these groups CALLs a different foreground and background color combination for the eight characters in character set 3. This new information is then sent to a subroutine beginning on Line 230.

Lines 230-320 define a subroutine that includes two FOR-NEXT loops. The first, Lines 270-290, PRINTs the characters represented by the A, B, and C variables four times. The second loop, Lines 300-310, delays the program's display by counting K a thousand times.

Line 320 RETURNs the program to the line after the one that initially directed the computer to the subroutine.

CALL GCHAR (row, column, numeric, variable)

The CALL GCHAR statement allows you to read the character code of a character located at a specific row and column number. If we enter the program

```
100 CALL CLEAR

110 CALL VCHAR(12,16,54,200)

120 CALL GCHAR(24,22,X)

130 PRINT X

140 CALL GCHAR(24,28,Y)

150 PRINT Y

160 GOTO 160
```

we see the familiar vertical columns of the number 6 (character code 54) and two numbers at the bottom left hand corner of the screen. The first CALL GCHAR at row 24 column 22 is within a space filled by the VCHAR statement with the number 6. Thus, when X is printed, we see a value of 54. The second GCHAR statement reads the character space at row 24, column 28. As you can see, this space is at the lower right hand corner and is blank. Hence, when Y is printed, we see the value for the blank space, 32.

THE RANDOM CHARACTER GENERATOR

The following program shows off the graphics statements we have reviewed with a dazzling and ever-changing display. The program asks you to enter a value within a certain range for all the terms accompanying the CALL COLOR, CALL SCREEN, CALL VCHAR, CALL HCHAR, CALL CHAR, and CALL GCHAR statements. Although you cannot choose values outside the legitimate range of these statements, you may include one, a few, or all the values for each term involved. The program then randomizes the values you have given, staying within your specified limits. If you enter small numbers the range of values the program may select from will be small and the display will be relatively simple. Using them all will produce entertaining complexity. Experiment with different values will get a feeling for how the different graphics statements interact and for your computer's overall graphics capability.

Something to keep in mind: when you CALL GCHAR, you affect the overall display. This is because to make the numbers of the GCHAR display readable, we must use CALL COLOR to change character sets 3 and 4 (which contain the numerals 0 to 9) to black on white; other

The CALL KEY statement at Line 420 allows you to interrupt and restart the program with a new set of values by pressing any key. The CALL COLOR statements at the end of the program then reset the screen to black on yellow, so that you can clearly see the INPUT requests.

```
100 CALL CLEAR
110 INPUT "COLOR SETS?(1 TO
16)":I
120 IF (I<1)+(I>16)THEN 110
130 INPUT "FOREGROUND COLORS
?(1 TO 16)":J
140 IF (J<1)+(J>16)THEN 130
150 INPUT "BACKGROUND COLORS
?(1 TO 16)":K
160 IF (K<1)+(K>16)THEN 150
170 INPUT "SCREEN COLORS?(1
TO 16)":L
180 IF (L<1)+(L>16)THEN 170
190 INPUT "NUMBER OF ROWS?(1
 TO 24)":M
200 IF (M<1)+(M>24)THEN 190
210 INPUT "NUMBER OF COLUMNS
?(1 TO 32) ":N
220 IF (N<1)+(N>32)THEN 210
230 INPUT "NUMBER OF CHARACT
ERS?       (0 TO 94)":O
240 IF (O<0)+(O>94)THEN 230
```

```
250 INPUT "CHARACTER REPETIT
IONS?       (1 TO 768)":P
260 IF (P<1)+(P>768)THEN 250
270 PRINT "DISPLAY GCHAR?(Y
OR N)"
280 CALL KEY(3,X,Y)
290 IF Y=0 THEN 280
300 CALL CLEAR
310 GOSUB 440
320 CALL COLOR(A,B,C)
330 CALL SCREEN(D)
340 CALL VCHAR(E,F,G,H)
350 CALL HCHAR (E,F,G,H)
360 IF X=89 THEN 380
370 GOTO 420
380 CALL GCHAR(E,F,T)
390 CALL COLOR(3,2,16)
400 CALL COLOR(4,2,16)
410 PRINT T
420 CALL KEY(3,XX,Y)
430 IF Y=0 THEN 310 ELSE 540
440 RANDOMIZE
450 A=INT(I*RND)+1
460 B=INT(J*RND)+1
470 C=INT(K*RND)+1
480 D=INT(L*RND)+1
490 E=INT(M*RND)+1
500 F=INT(N*RND)+1
510 G=INT(O*RND)+32
520 H=INT(P*RND)+1
530 RETURN
540 CALL CLEAR
550 FOR SET=1 TO 8
560 CALL COLOR(SET,2,1
570 NEXT SET
580 CALL SCREEN(11)
590 GOTO 110
```

Lines 110-260 ask you to select a range of values for each of the graphics commands. Each INPUT statement is followed by an IF-THEN statement that checks if the INPUT value is appropriate. If not, the IF-THEN statement sends the computer back to the INPUT line for a new value. These variables represent the range of values from which a single value will later be randomly chosen for each aspect of the program. Entering 16 when the computer asks for COLOR SETS? thus gives the program colors sets 1 through 16 to choose from.

The table below lists the variable names and which part of the program they affect. The column on the right lists the variables that will later be associated with the INPUT variables in the subroutine in Lines 44Ø-53Ø. This last set of variables will be used to randomly choose a value for each graphics feature.

Program Aspect	INPUT Variable	Randomizer Variable
Color Sets	I	A
Foreground Colors	J	B
Background Colors	K	C
Screen Colors	L	D
Number of Rows	M	E
Number of Columns	N	F
Number of Characters	O	G
Character Repetitions	P	H

Lines 27Ø-29Ø let you decide whether to display the results of a CALL GCHAR statement. CALL KEY provides this information (by CALLing Y or N) after the DISPLAY GCHAR message has been printed. Line 29Ø sends the computer back to Line 28Ø if no key is pressed.

Line 31Ø starts the subroutine at Line 44Ø, which produces values for the "randomizer" variables A through H.

Lines 31Ø-35Ø comprise five graphics statements that are performed repeatedly. The subroutine in Lines 44Ø-53Ø determines the values of the variables in these statements. The action in this section of the program flows into Line 43Ø where the computer is directed to the GOSUB statement in Line 31Ø if a key has not been pressed. This GOSUB statement then recalls the subrountine in Lines 44Ø-53Ø which generates a new set of randomizer variables for each graphics statement. Thus the statements in this section are performed again and again, each time with a new set of variable values.

Lines 36Ø-37Ø The variable corresponding to the character code for the keyboard keys (defined in the CALL KEY statement in Line 28Ø) is used to determine whether to display GCHAR. If a "Y" for YES is pressed, the value of 89 is assigned to the variable X. In this case, the program skips over Line 37Ø and performs the GCHAR routine. If X does not equal 89, the computer moves to Line 37Ø, which tells it to skip over the GCHAR routine.

Lines 38Ø-41Ø detail the routine that prints CALL GCHAR results on the screen. The CALL GCHAR statement in Line 38Ø is followed by statements setting the color for character sets 2 and 3, which contain the number characters. Without these statements you might not be able to read the results of the GCHAR statement. But the screen display is also affected, since other characters already on the screen that happen to belong in character sets 2 and 3 will also change. After the colors for these sets have been set to black on white, the results of GCHAR are printed at Line 41Ø.

Lines 42Ø-43Ø is the CALL KEY statement that lets you restart the program for a new set of INPUT variable values. If you simply let the program run without pressing any key, the program action goes back to Line 31Ø to fetch the subroutine for the randomizer variables, as before. If you do press a key, the action moves moves to Line 54Ø, which prepares the screen to restart the program.

Lines 44Ø-53Ø contain the subroutine that defines variables A through H for supplying randomized values to the variables named in the graphics statements in Lines 32Ø to 35Ø. The names of the INPUT variable associated with each of these new randomized variables are enclosed within the parentheses of each random number function. The random number function multiplies each of these INPUT variables by a randomly assigned value between one and zero. One is then added to ensure that the result does not equal zero. Each of these statements picks a single value randomly from the range of values specified in your INPUT. (Check the table that accompanies the explanation for Lines 11Ø-26Ø to see which of the original INPUT variables is associated with each of these new variables.) The RETURN statement returns program action to Line 32Ø where the graphics display begins.

Lines 54Ø-59Ø prepare the program to begin again. While the program RUNs, colors and character sets change continually. To start with a new set of INPUT values, we need a uniform color scheme. After CALLing the screen CLEAR, a FOR-NEXT loop calls a foreground color of black and a background color of dark yellow for all the characters in the first eight character sets. Then Line 58Ø also turns screen color dark yellow, and Line 59Ø returns us to the beginning.

COMBINING CHARACTERS IN SPACE: WHITE KNIGHT

So far, we have looked only at creating characters in a single character space, but we can also create larger images by piecing character spaces together, much like a jigsaw puzzle. The following program demonstrates this technique.

A chess piece, a knight, is constructed out of four character spaces near the center of the screen. You are then asked to select foreground and background colors for this knight. Enter values from one to sixteen for your colors. Press any key before choosing a new set of colors.

Notice the number of the row and column numbers in the HCHAR statements. This spacing is graphically demonstrated by the technique of using the character codes for the numbers 1 through 4, which have been reassigned to represent each of the spaces making up the knight. Because of this, you will see all of your INPUT numbers in the same colors as the knight since they belong to the same character set which determines the knight's colors. The numbers 1 to 4 will appear as pieces of the knight itself since the character codes reserved for these numbers have been redefined to make the knight. Try entering a value of 12 for the foreground color and one of 34 for the background color. The program will BREAK because 34 is a bad color value, but the pieces of the puzzle will be revealed!

```
100 CALL CLEAR
110 CALL CHAR(49,"0000010608
121021")
120 CALL CHAR(50,"000000C020
100808")
130 CALL CHAR(51,"2618010204
0F0000")
140 CALL CHAR(52,"8C84040408
F80000")
150 CALL HCHAR(9,15,49)
160 CALL HCHAR(9,16,50)
170 CALL HCHAR(10,15,51)
180 CALL HCHAR(10,16,52)
190 INPUT "FOREGROUND COLOR?
":F
200 INPUT "BACKGROUND COLOR?
":B
210 CALL COLOR(3,F,B)
220 CALL SCREEN(11)
230 CALL KEY(3,X,Y)
240 IF Y=0 THEN 230 ELSE 110
```

COMBINING CHARACTERS IN TIME: RUNNING MAN

Now let's look at a program that really *RUNS*. It uses four characters like the previous one, but in a different way. Here only one character is displayed at any given moment. It is the character's relationship in time, rather than space, that we are interested in. Creating smooth movement in TI BASIC is difficult, because we can only display characters block by block. To improve matters somewhat, this program displays two characters for each character space—one that occupies either the left or right side of a character space. Thus although four images are produced, our running man only moves two spaces.

In Lines 12Ø to 15Ø, we define the four characters which, when linked together, produce the illusion of motion. Characters 13Ø and 131 are displayed in the same place, and characters 132 and 133 are similarly paired. Each time character 131 is displayed, the computer moves the column location in the COL variable one step to the left on the screen. This also happens after character 133 is displayed. Between CALLing CHARacters, we have inserted a CALL CLEAR statement to erase the previous character before the next one is displayed. Thus, we create an impression of smooth motion. The process is analogous to passing movie frames one at a time through a projector.

You can control the running man's speed with the INPUT statement in Line 16Ø. FOR-NEXT loops in Lines 21Ø-22Ø, 25Ø-26Ø, 3ØØ-31Ø, and 34Ø-35Ø create delays between character displays. By assigning a low value to the variable X, you can make the runner run fast. To see each of our runners' positions clearly, enter a delay of 1ØØ.

```
1ØØ CALL CLEAR
11Ø CALL SCREEN(11)
12Ø CALL CHAR(13Ø,"Ø4ØØ14ØEØ
5ØC1221")
13Ø CALL CHAR(131,"2ØØØ2ØF82
Ø2Ø5848")
14Ø CALL CHAR(132,"Ø4ØØØ5ØE1
4Ø5ØAØ4")
15Ø CALL CHAR(133,"2ØØØ2ØF82
Ø2Ø584Ø")
16Ø INPUT "DELAY?(PRESS Ø FO
R NO DELAY)":X
17Ø COL=32
18Ø Y=1
19Ø COL=COL-Y
2ØØ CALL HCHAR(15,COL,13Ø)
```

```
210 FOR DELAY=1 TO X
220 NEXT DELAY
230 CALL CLEAR
240 CALL HCHAR(15,COL,131)
250 FOR DELAY=1 TO X
260 NEXT DELAY
270 COL=COL-Y
280 CALL CLEAR.
290 CALL HCHAR(15,COL,132)
300 FOR DELAY=1 TO X
310 NEXT DELAY
320 CALL CLEAR
330 CALL HCHAR(15,COL,133)
340 FOR DELAY=1 TO X
350 NEXT DELAY
360 CALL CLEAR
370 IF COL<=2 THEN 160 ELSE
190
```

16 Interacting with Your Computer: Keyboard and Joystick

Two of the most commonly used devices for entering information into a program are the keyboard and joysticks. The keyboard, strictly speaking, is not actually part of your computer but a built-in peripheral device used to communicate with it. We are all aware that the keyboard can be used to create programs to run on the computer. It is also true that by using the CALL KEY command in TI BASIC, the keyboard can be used to place information into a program while it is running. In this chapter we will investigate the use of the CALL KEY command for creating keyboard program interaction and the CALL JOYST command which controls the use of joysticks.

CALL KEYBOARD

The CALL KEY statement works something like the INPUT statement: both allow you to enter data into a program while it is running. But unlike INPUT, CALL KEY does not disturb the screen display, a real advantage for graphics programs.

The CALL KEY statement has three terms: a key-unit, a return variable, and a status variable. The correct form for a CALL KEY statement is:

```
200 CALL KEY(3,X,Y)
```

where "3" is the **key-unit,** "X" the **return variable,** and "Y" the **status variable.**

Key-units

Key-units, from 0 to 5, define specific keyboard scanning modes that tell the computer which values are placed in the return variable when keys are pressed. Depending on which key-unit you specify, the computer refers to a different keyboard map to determine these values. Pages III-3,4 of your *User's Reference Guide* shows diagrams of the keyboard maps corresponding to each key-unit.

Key-unit 0: keeps the keyboard map called for in a previous CALL KEY statement.

Key-units 1 and 2: correspond to the left and right halves of the keyboard and are used when you need two sources of CALL KEY information, as in a two-player game. When two CALL KEY statements with these key-units are used, the keyboard is split into two independent input devices. These key-units may also be used to establish separate sources of control for two joysticks.

Key-unit 3: the standard 99/4A keyboard mode. In this mode values for upper- and lower-case letters are returns as upper-case letters. Values from 1 to 15 are returned for functions like INSert, DELete, etc.

Key-unit 4: maps the keyboard for use with the Pascal language.

Key-unit 5: the TI BASIC key-unit. The computer generates separate values for both upper- and lower-case letters as well as for 15 function (FCTN) keys and 32 control (CTRL) keys.

Return Variable

CALL KEY sends the computer off to its keyboard maps to find the character codes corresponding to the ASCII (American Standard Code for Information Interchange) character code for whatever key you press. The computer then returns that character code as the value of this next CALL KEY term, the return variable. For example, the character code for the "1" key is 49.

Refer to the list of character codes in the Appendix of your *User's Reference Guide* as you try the following exercise:

```
100 CALL KEY(5,X,Y)

110 PRINT X

120 GOTO 100
```

As the program runs, you'll see a string of −1s scrolling up the screen: −1 is the value assigned to the return variable X if no key has been pressed. Now try pressing some keys and see what happens. The numbers PRINTed by the computer should correspond to the character codes of the keys you press.

If you change the key-unit term to 3 you'll find that the codes for lower-case letters are the same as those for upper-case letters. Values returned by FCTN and CTRL keys are outside of the normal range of values returned by your keyboard's symbols.

Status Variable

This term of the CALL KEY statement variable tells the computer what the status of the keyboard action is. There are three possible values for the status variable:

status =	0	No key has been pressed.
status =	1	A new key has been pressed.
status =	−1	The same key has been pressed.

Change Line 110 in the above program to PRINT Y. The computer prints a series of zeros until you touch a key. If you touch a key briefly, you get a "1" and then more zeros. If you hold a key down, you get a "1" and then some "−1's" until you let up on the key.

The CALL KEY statement gives you a way to stop program action until you're ready to proceed. Programs with several pages of initial instructions often use CALL KEY to hold text on the screen until a PRESS ANY KEY TO CONTINUE direction is performed. Setting this up simply requires telling the computer to GOTO the CALL KEY statement IF the status variable equals 0 (no key pressed).

The next program demonstrates how the return and status variables interact by printing messages based on the values of these variables when different keys are pressed. When you run the program, the message PRESS ANY KEY TO CONTINUE appears. As soon as you press a key, the character you have pressed appears in a sentence giving its character code. If you hold down a particular key for more than a moment, another message appears and repeats itself for as long as you hold the key down.

```
100 CALL CLEAR
110 PRINT " PRESS ANY KEY TO
 CONTINUE"::
120 CALL KEY(5,R,S)
130 IF S=0 THEN 120
140 IF S=1 THEN 150 ELSE 160
150 PRINT "THE ASCII CODE FO
R ";CHR$(R);" IS";R
160 IF S=-1 THEN 170 ELSE 12
0
170 PRINT "  ";CHR$(R);"'s C
ODE IS STILL";R;"!!"
180 GOTO 120
```

Experiment with the ALPHA LOCK key up (off) and press keys all over the keyboard to see what happens. (Watch out for FCTN:CLEAR, which will interrupt the program, and FCTN:QUIT, which will obliterate everything and send you back to the TI start-up screen.) The key-unit here is 5, so upper- and lower-case letters have separate codes. If the ALPHA LOCK key is down, however, you'll only see upper case, regardless of the position of the SHIFT key. Key-unit 5 also returns values for 15 different FCTN combinations and 32 CTRL combinations. Those strange characters that appear when you use the CTRL keys come from within the computer's memory, but are not necessarily part of the program. You can define characters which are in the range of ASCII code values returned by the CTRL and FCTN keys, however, and even enter these characters directly into PRINT and DATA statements in your programs by using these keys!

This program's logic goes like this:

Lines 100-130 set the initial screen and keep it in place until you do something. Line 130 tells the computer to keep returning to the CALL KEY in 120 as long as the status variable S equals zero, or NO KEY PRESSED.

Lines 140-150 Line 140 directs the computer to go to Line 150 if the status variable equals 1, or NEW KEY PRESSED; Line 150 then prints its message.

Lines 160-180 IF the status variable now equals −1 or SAME KEY PRESSED, the program proceeds to Line 170 and prints its message. Line 180 loops back to the CALL KEY statement in Line 120.

Note: The CHR$ in Lines 150 and 170 is a string function that allows the program to print the character of each key pressed while the program is running through the print statements on these two lines. CHR$

produces the character whose ASCII code is R, the same as the return variable R. We'll cover string functions in Chapter 18, "String Functions."

CALL KEY can accompany other statements in TI BASIC. The next program is very similar to the one demonstrating the ON-GOTO statement in Chapter 9. But here we'll combine the ON-GOTO statement with CALL KEY instead of INPUT.

First we have to convert the character code values of CALL KEY's return variable into values that will allow the ON-GOTO statement to choose among several line numbers. As you may remember, a value of 2 for X in ON X GOTO 16Ø, 18Ø, 2ØØ, 22Ø, etc., transfers the program activity to Line 18Ø because 18Ø is the second line number listed in the ON-GOTO statement.

Lines 13Ø and 14Ø of our new program perform and control this transition from the CALL KEY statement to the ON-GOTO statement. Line 13Ø limits the acceptable values in the CALL KEY return variable to 49, 50, 51, and 52—the character codes for the numbers 1, 2, 3, and 4, respectively. Line 14Ø then subtracts the number 48 from these code values, actually converting them to 1, 2. 3, and 4. The ON-GOTO statement in Line 15Ø then picks one of its four destination line numbers, depending on which key was pushed.

```
1ØØ CALL CLEAR
11Ø PRINT "BATTER'S UP! HOW
MANY BASES WILL YOU GET?(1 T
O 4)"
12Ø CALL KEY(3,X,Y)
13Ø IF (X<49)+(X>52)THEN 12Ø
14Ø X=X-48
15Ø ON X GOTO 16Ø,18Ø,2ØØ,22
Ø
16Ø PRINT "A SINGLE!"
17Ø GOTO 12Ø
18Ø PRINT "A DOUBLE!"
19Ø GOTO 12Ø
2ØØ PRINT "A TRIPLE!"
21Ø GOTO 12Ø
22Ø PRINT "HOME RUN!"
23Ø GOTO 12Ø
```

Mazemaker

Now let's see how the information CALL KEY puts into a program can control program flow and create interaction between you and the program.

Starting at the center of a black screen, a white block appears. By pushing the keys E, X, F, and S (marked with arrows), you may move this block up, down, right, and left. If you go off the screen, the block reappears on the other side at the same row or column location. By pressing FCTN and one of those letters (as if you were using the editor), you can erase blocks from the screen.

To get these effects, we need two characters—one to draw blocks and another to erase them. Pressing FCTN first gives each of the four letters a second code. The eight codes can then be interpreted in CALL KEY's return variable as alternative actions, drawing and erasing.

```
100 CALL CLEAR
110 CALL CHAR(131,"00FEFEFEF
EFEFEFE")
120 CALL CHAR(132,"0000000000
00000000")
130 X=12
140 Y=16
150 CALL COLOR(13,16,2)
160 CALL SCREEN(2)
170 CALL V CHAR(X,Y,131)
180 CALL KEY(3,F,G)
190 IF G=0 THEN 180
200 IF F=69 THEN 310
210 IF F=11 THEN 310
220 IF F=88 THEN 400
230 IF F=10 THEN 400
240 IF F=83 THEN 490
250 IF F=8 THEN 490
260 IF F=68 THEN 580
270 IF F=9 THEN 580
280 IF F=32 THEN 100
290 GOTO 180
300 REM ***MOVING UP!***
310 X=X-1
320 IF X<1 THEN 330 ELSE 340
330 X=24
340 IF F=11 THEN 370
350 CALL VCHAR(X,Y,131)
360 GOTO 180
370 CALL VCHAR(X,Y,132)
380 GOTO 180
390 REM ***MOVING DOWN!***
400 X=X+1
410 IF X>24 THEN 420 ELSE 43
0
420 X=1
430 IF F=10 THEN 460
440 CALL VCHAR(X,Y,131)
```

```
450 GOTO 180
460 CALL VCHAR(X,Y,132)
470 GOTO 180
480 REM ***MOVING LEFT!***
490 Y=Y-1
500 IF Y<1 THEN 510 ELSE 520
510 Y=32
520 IF F=8 THEN 550
530 CALL VCHAR(X,Y,131)
540 GOTO 180
550 CALL VCHAR(X,Y,132)
560 GOTO 180
570 REM ***MOVING RIGHT!***
580 Y=Y+1
590 IF Y>32 THEN 600 ELSE 61
0
600 Y=1
610 IF F=9 THEN 640
620 CALL VCHAR(X,Y,131)
630 GOTO 180
640 CALL VCHAR(X,Y,132)
650 GOTO 180
```

Lines 100-170 set up the screen and program information before you touch a key. The draw and erase characters are defined; the screen is made black, the character colors are set, the square is located and displayed in the center of the screen.

Lines 180-290 contain the CALL KEY statement to which the program keeps returning. Lines 190 to 280 "decide" what to do depending on what happens at the keyboard. Line 190 tells the computer to return to the CALL KEY in Line 180 if no key is pressed. Lines 200 to 280 give instructions for the eight different key codes corresponding to the four draw keys and the four erase keys. Notice that each draw key code is paired with an erase key code that sends the computer to the same line number. "Draw" return variable values are 69, 88, 83, and 68. When you press FCTN first (to erase), the codes returned are 11, 10, 8, and 9.

Lines 300-380 move the draw and erase characters toward the top of the screen. Line 310 determines the direction by making each new row number one less than the previous one. Lines 320 and 330 prevent bad values in the VCHAR statement (there is no row 0!) by sending the character back to the bottom of the screen once it has gone off the top. Line 340 sends the computer to the erase routine on Line 370 IF FCTN:↑ has been pushed. Notice that unless this condition is fulfilled,

the program skips the erase routine by returning to the CALL KEY in Line 360. Only the draw *or* the erase routine can occur.

Lines 390-470, 480-560, 570-650 do exactly the same thing for the downward, left, and right motions that Lines 300-380 do for the upward motion. At the end of each routine, the computer returns to the CALL KEY statement for further instructions.

CALL JOYSTICK

Like CALL KEY, the CALL JOYST statement lets people interact with programs. CALL KEY makes sense when you want to "talk" to the computer in words of numbers; CALL JOYST, on the other hand, is ideal for moving images around on the screen and playing games.

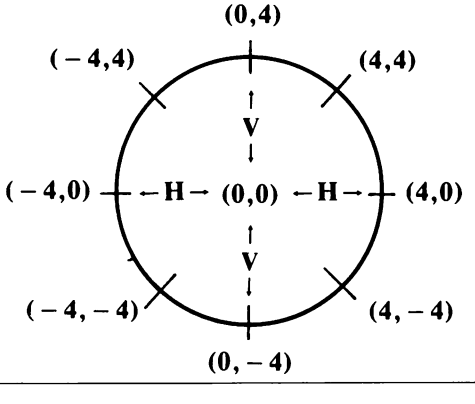
The CALL JOYST statement uses key-units 1 and 2 to split the keyboard in two for two different sets of input—one for each joystick in a two-person game, for example.

Joysticks have two sources of information: the stick itself and the "fire" button. The fire button is actually a key requiring a CALL KEY statement with either key-unit 1 or 2, depending on which key-unit accompanies CALL JOYST. The character code for the return variable of this CALL KEY statement—whether key-unit 1 or 2, is 18.

The other source of input is peculiar to the joystick itself, which is CALLed in a CALL JOYST statement. Fully punctuated, CALL JOYST might look like this in a program:

```
200 CALL JOYST(1,H,V)
```

where the 1 is the key-unit, H is the horizontal movement value and the V is the vertical movement value. There are nine possible combinations of horizontal and vertical values, all derived from combinations of the integers 4, 0, and −4. This arrangement, including diagonal movements and the at-rest position, appears in the following diagram.



You must manipulate these sets of values in programs to get results from the CALL JOYST statement.

To see how joystick input works, try this short program. NOTE: joysticks work imperfectly when the ALPHA LOCK key is down; be sure to leave it up when running joystick programs!

```
100 CALL KEY (1,R,S)

110 PRINT ,R;S

120 CALL JOYST (1,H,V)

130 PRINT H;V

140 GOTO 100
```

Four columns of figures will appear on the screen. The two columns on the left half of the screen are the horizontal and vertical values generated by moving the joystick. The two columns on the right are the return and status variables of the CALL KEY statement. Experiment with the joystick circle and the fire button and see what values you get.

Joystick Mazemaker

This program is analogous to the CALL KEY Mazemaker but is built around the CALL JOYST statement. You can see that this version is shorter and gives you more directions to move in, including four diagonals. Note that of the character blinks on and off when you push fire button. To start a new maze, press the letter S.

```
100 CALL CLEAR
110 CALL CHAR(36,"183C7EFFFF
7E3C18")
120 CALL SCREEN(2)
130 CALL COLOR(1,16,2)
140 A=16
150 B=12
160 CALL JOYST(1,H,V)
170 A=A+H/4
180 B=B-V/4
190 IF A<1 THEN 200 ELSE 210
200 A=32
210 IF B<1 THEN 220 ELSE 230
220 B=24
230 IF A>32 THEN 240 ELSE 25
0
240 A=1
250 IF B>24 THEN 260 ELSE 27
0
```

```
260 B=1
270 CALL KEY(1,R,S)
280 IF R=2 THEN 100
290 IF R=18 THEN 300 ELSE 34
0
300 CALL CHAR(40,"FFFFFFFFFF
FFFFFF")
310 CALL COLOR(2,2,2)
320 CALL HCHAR(B,A,40)
330 GOTO 160
340 CALL HCHAR(B,A,36)
350 GOTO 160
```

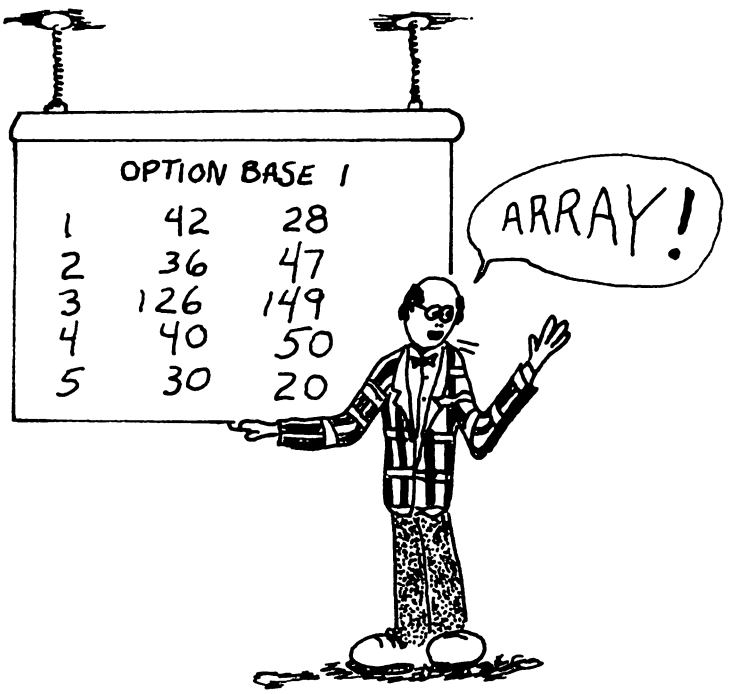
Lines 100-150 define the shape and color of character 36, CALL the SCREEN black, and place the character in the center of the screen (row 16, column 12).

Lines 160-180 The program revolves around the CALL JOYST statement in Line 160. These lines define the movement of the character—one space in any direction at a time—by dividing the normal joystick input values of 4 and −4 by 4 to get 1 or −1.

Lines 190-260 prevent bad values in the CALL HCHAR statements in Lines 320 and 340. (There can be no rows higher than the 24th or lower than the first, or columns higher than the 32nd or lower than the first.) These statements make the character to "wrap around," reappearing on the opposite side of the screen.

Lines 270-350 relate to the CALL KEY statement on Line 270. Line 280 returns program action to the beginning to clear the screen if an S has been pressed. If the fire button is being pressed, Line 290 sends the action to 300-310 to produce a character which erases the white character on the screen by covering it with a black one. If the joystick fire button is not being pressed (and R therefore does not equal 18), the ELSE 340 in Line 290 sends the program on to draw another white character. Notice that only the erase *OR* the drawing character may be used, because of the combination of the IF-THEN-ELSE statement in Line 290 and the GOTO in Line 330, which separates the two actions.

17 Arrays



In Chapter 10 we saw how READ/DATA statements control the flow of variable information within a program. Although these statements are often useful, they have certain drawbacks.

For one thing, READ/DATA statements allow only limited DATA maniuplation. We can either RESTORE the entire DATA list or RESTORE part of the list by specifying a particular DATA line number. But what if we want to re-use a single item of DATA without the rest of the items on the same line? Another drawback is that the computer always READs DATA sequentially. There's no way to jump around the DATA list and select individual pieces of information because the computer has no way to distinguish one piece of DATA from another. Finally, the DATA must be arranged so that the program runs successfully; we

end up with long and cumbersome DATA lists because we are limited in the ways we can restructure the DATA while the program is running.

Fortunately, we can get around these problems using something called an **array.**

Essentially, an array is nothing more than a list. What makes it different from a list in a DATA statment, however, is that in creating an array, we set aside a portion of the computer's memory to store the list. Once an array is stored in memory, we can access it as many times as we like without having to RESTORE the information.

Arrays differ from lists, however, in that lists have only one dimension, a single column of numbers. Arrays may have several dimensions. For example; picture five rows of six numbers each: what you'd have is a two-dimensional array. Each number's position in your array can be pinpointed by its row and column number, something like locating Paris by its latitude and longitude.

SUBSCRIPTED VARIABLES AND SIMPLE ARRAYS

This simple program uses an array to change the screen color four times. There are easier ways of doing this, to be sure, but the program clearly demonstrates certain properties of arrays.

```
100 SC(0)=5
110 SC(1)=7
120 SC(2)=13
130 SC(3)=16
140 CALL SCREEN(SC(I))
150 FOR DELAY=1 TO 200
160 NEXT DELAY
170 I=I+1
180 IF I=4 THEN 200
190 GOTO 140
200 END
```

We create our array in Lines 100-130. It consists of four parts, or elements, called **subscripted variables.** They are SC(0), SC(1), SC(2), and SC(3). Part of the name—SC, for screen color—remains the same; the other part—the subscript—varies. Thus, we have an array for SC, or screen color, with individual elements, which are the colors.

When the computer encounters a subscripted variable such as SC(0), it recognizes it as an array and immediately sets aside, or, **dimensions** a portion of memory that can hold up to 11 elements. So even though we only used four elements in our array, enough memory was

automatically reserved to hold SC(4)-SC(1Ø). If you wish to create an array with more than 11 elements, you need another array-devising procedure that we'll describe shortly. This procedure requires the DIM statement.

When creating an array with subscripted variables, the first character in the name must be a letter. A second (optional) character may be a number or a letter. Thus, SC(Ø), S5(Ø), and S(Ø) are all possible choices; 3S(Ø), 33(Ø), and 3(Ø) are syntactically illegal.

Lines 1ØØ-13Ø also assign the information to be held in the array, much like assigning values to variables. Here we use a direct method of assignment by simply stating SC(Ø) = 5, SC(1) = 7 etc. The computer places that information into memory, storing 5 as the first element on the SC list, 7 as the second, and so on. Because zero is a perfectly good quantity to the computer, the first element of an array receives the number Ø, while the second element receives the number 1, and so on.

Line 14Ø tells the computer to utilize this information each time it passes through the loop between Lines 14Ø and 19Ø. The loop consists of the CALL SCREEN (SC(I)) statement, a FOR-NEXT loop that keeps the screen color on for a moment, a counter to keep track of the elements in the array; and an IF-THEN statement to signal the loops' end. Finally, the GOTO 14Ø statement on Line 19Ø creates the loop.

The first time through the loop, I will be Ø, so the computer will interpret SC(I) as SC(Ø) and will respond by retrieving the piece of information (5) that has been stored under the name SC(Ø). The second time through the loop, I will be 1, so SC(1) will yield its value (7) and so on. When I = 4, the IF-THEN statement will become true, and the computer will leave the loop and END the program.

To see this more clearly, add the following line to your program:

```
145 PRINT I;SC(I)
```

When the program RUNs, the screen color changes; and as it does, the subscript and value of each element in the array is PRINTed.

To verify that the array is indeed stored in memory or to check that a particular element in the array is correct, try this little trick. Without using a program line, simply type in: `PRINT SC(2)+SC(1)`

You should receive the answer 20 because SC(2) = 13 and SC(1) = 7.

You can get the same display with FOR-NEXT loops instead of counters:

```
100 SC(0)=5
110 SC(1)=7
120 SC(2)=13
130 SC(3)=16
140 FOR I=0 TO 3
150 CALL SCREEN(SC(I))
160 FOR DELAY=1 TO 200
170 NEXT DELAY
180 NEXT I
```

USING READ/DATA TO LOAD THE ARRAY

Besides direct assignment, we can also READ a DATA list into an array. The diffference between the DATA READ into an array and a regular DATA list is that the DATA is now stored in memory, and each element of the list has been given a name. Let's write our program for changing screen color once again, only this time we'll load the array using the READ/DATA statements.

```
100 FOR I=0 TO 3
110 READ SC(I)
120 NEXT I
130 FOR I=0 TO 3
140 CALL SCREEN(SC(I))
150 FOR DELAY=1 TO 200
160 NEXT DELAY
170 NEXT I
180 DATA 5,7,13,16
```

In this program, the first I loop loads the array; the second CALLs the screen colors. The nested DELAY loop keeps the color on the screen.

After the program is run, without using a line number, type:

```
PRINT SC(0);SC(1);SC(2);SC(3)
```

The computer will print out the values stored in the array under those names.

OPTION BASE 1 AND THE DIM STATEMENTS

When a subscripted variable is introduced into a program, enough memory is set aside to hold 11 elements; in our sample programs, we've called them SC(0)-SC(10). If however, you wish to store more than 11 elements in an array, you must use the DIMension statement.

The DIM statement instructs the computer to set aside, or "DIMension," enough memory to hold a list. Thus, DIM SC(25) would DIMension an array to hold 26 elements: SC(Ø)-SC(25). If we do not DIMension enough memory to hold our list, we'll get an error when the program runs.

The DIM statement must appear in a program before any mention of the array itself; otherwise the memory space will not be set aside in time. For this reason, it's best to DIMension any arrays at the beginning of a program.

OPTION BASE(1) is a TI BASIC statement that allows you to set the lower limit of the array subscript at 1 instead of the usual 0. If you find the concept of using a zero as the first element in the array and a one as the second element too much of a double-think, then use the OPTION BASE 1 statement. This statement must appear before the DIM statement and will affect all arrays in the program. No array can contain a 0 subscript if you use OPTION BASE 1.

The following program demonstrates all the array concepts covered thus far. When you RUN the program for the first time, it will return the following screen display.

```
*
**
***
****
*****
******
*******
******
*****
****
***
**
*
INPUT NEW SUBSCRIPT (1-13)
?
```

By entering a number from 1 through 13, you can change the subscript and get a new graphics display.

```
100 CALL CLEAR
110 OPTION BASE 1
120 I=1
130 DIM F$(20)
140 READ F$(I)
150 IF F$(I)="A" THEN 180
160 I=I+1
170 GOTO 140
180 I=1
190 IF F$(I)="A" THEN 240
200 PRINT F$(I)
210 I=I+1
220 GOTO 190
230 DATA *,**,***,****,*****
,******,*******,******,*****
,****,***,**,*,A
240 PRINT "INPUT NEW SUBSCRI
PT (1-13)"
250 INPUT I
260 GOTO 190
```

Line 100 clears the screen.

Line 110 instructs the computer to use 1 as the first subscript.

Line 120 initializes the array counter to 1 to coincide with the OPTION BASE 1.

Line 130 DIMensions an array that can hold up to 20 elements.

Line 140 begins the array loading procedure by READing the first element in the DATA list and assigning it to the first element in the array.

Line 150 tests to see if the FLAG "A" is READ. IF it is, THEN the computer stops READing DATA and goes to Line 180.

Line 160 increments the array counter I by 1.

Line 170 creates the array-loading loop by sending the computer back to Line 140 to READ the next element into the array.

Line 180 resets the array counter I to 1 for the PRINT portion of the program.

Line 190 checks for the Flag A. IF the computer encounters the flag, THEN it leaves the PRINT loop and goes to Line 240.

Line 200 PRINTs the current value of F$(I).

Line 210 increments the array counter I by 1.

Line 220 creates the PRINT loop, by transferring control back to Line 190, which will check for the flag before PRINTing at Line 200.

Line 230 contains the DATA to be READ into the array.

Line 240 contains a prompt asking for a new subscript.

Line 25Ø inputs the new subscript into the program, reintializing the array counter I.

Line 26Ø returns to the PRINT loop.

Although we used only 14 pieces of DATA, or elements, we DIMensioned our array to hold 20. If you're not sure how many elements you need for an array, you can set aside more memory than you expect to use. If you start running out of memory, you can re-DIM the array later.

By placing a flag in the DATA, we can end the loading process without having to specifiy how many elements are to be loaded. We can also use this flag again in the PRINT portion of the program as a signal to leave the PRINT loop.

Try INPUTting a few new subscripts. As the OPTION BASE 1 makes the number 1 the first element of the array and the number 4 the fourth element, keeping track of the array becomes an easy process.

You might also try to enter a value larger than 14. The computer will then return a BAD SUBSCRIPT IN 19Ø error message. This is because you will have tried to enter a value that has not been stored in the array.

TWO-DIMENSIONAL ARRAYS

We are now going to add the flexibility of two-dimensional arrays to our repertoire of programming techniques. The following program stores nine sets of related numbers in a two-dimensional array. Each set has three variables, representing the duration and frequencies of two notes in a CALL SOUND statement. The program plays a scale four times. The first time, it plays the entire scale. On the second pass, it begins at the third note of the scale. On the third pass, it begins at the fifth note, and on the final pass, it again plays the whole scale. (You don't have to understand music to understand how the array works in this program.)

```
100 REM ***LOAD THE ARRAY***
    **********************
110 OPTION BASE 1
120 DIM V(10,3)
130 Z=1
140 READ DR,FQ1,FQ2
150 V(Z,1)=DR
160 V(Z,2)=FQ1
170 V(Z,3)=FQ2
180 Z=Z+1
190 IF DR=-1 THEN 220
200 GOTO 140
```

```
210 REM ***PLAY THE MUSIC***
    **********************
220 Z=1
230 IF V(Z,1)=-1 THEN 300
240 CALL SOUND(V(Z,1),V(Z,2)
,0,V(Z,3),2)
250 Z=Z+1
260 GOTO 230
270 DATA 200,262,659,200,294
,698,200,330,784,200,349,880
280 DATA 200,392,988,200,440
,1047,200,494,1175,600,523,1
319,-1,-1,-1
290 REM ***DATA CONTROL***
    **********************
300 C=C+1
310 ON C GOTO 320,340,360,38
0
320 Z=3
330 GOTO 230
340 Z=5
350 GOTO 230
360 Z=1
370 GOTO 230
380 END
```

We've divided the program into three parts. The first part READs the DATA for note duration and frequencies into a two-dimensional array. Let's take a closer look at how this array has been loaded.

Line 110 instructs the computer to use OPTION BASE 1.

Line 120 DIMensions enough memory to hold an array with three groups of information, each group holding a maximum of 10 elements.

Line 130 sets up the array counter Z, which keeps track of the elements being loaded. Since we are using OPTION BASE 1, the counter starts at 1.

Line 140 READs the first three pieces of DATA from the DATA bank and assigns them the names DR (duration), FQ1 (frequency for Voice 1), and FQ2 (frequency for Voice 2).

Line 150 assigns the variable DR to the first group, or table, of the array V(Z,1). The first time through the loop, the first piece of DATA (200), is assigned to the array location V(1,1).

Line 160 assigns the variable FQ1 to the second table of the array V(Z,2). The first time through the loop, the second piece of DATA (262), is assigned to the array location V(1,2).

Line 17Ø assigns the variable FQ2 to the third table of the array V(Z,3). The first time through the loop, the third piece of DATA (659), is assigned to the array location V(1,3).

Line 18Ø increments the array counter Z by 1.

Line 19Ø checks the value of the variable DR. IF it equals −1, our flag, THEN the computer stops READing DATA and moves to another part of the program, Line 22Ø.

Line 2ØØ creates the array-loading loop, by sending the computer back to the READ statement in Line 14Ø.

In the statement DIM V(1Ø,3) 3 represents the three columns of information loaded off the DATA. To get a more graphic understanding of this concept, type in the line below. When added to your program, this line will actually allow you to watch as your array is being loaded.

```
175 PRINT Z;V(Z,1);V(Z,2);V(
Z,3)
```

Now, when you run your program, you will receive the following printout.

```
1   2ØØ   262   659
2   2ØØ   294   698
3   2ØØ   33Ø   784
4   2ØØ   349   88Ø
5   2ØØ   392   988
6   2ØØ   44Ø   1Ø47
7   2ØØ   494   1175
8   6ØØ   523   1319
9    -1    -1    -1
```

The first number represents the array element counter Z. The second represents the duration, while the third and fourth represent the frequencies for Voices 1 and 2.

If you type the following line:

```
PRINT V(4,2)
```

the computer will return 349, because the piece of DATA located in the array as the fourth element of the second table is 349.

One important note: if you were not using OPTION BASE 1, then the statement DIM V(1Ø,3) would DIMension an array that would hold four groups of DATA with eleven elements. This is because the computer would use zero as the first value.

The second part of our program (Lines 22Ø-25Ø) plays the music, using the information stored in the array V.

Line 22Ø sets the array element counter Z to 1 since, at this point, it has been incremented to 10.

Line 230 checks the current value of V(Z,1), the duration. IF it equals −1, THEN the computer moves on to Line 3ØØ.

Line 24Ø CALLs the sound. The first time through the loop, Z = 1, so the sound statement asks for duration, located in the array as V(1,1). It also asks for frequency values, encoded as V(1,2) for Voice 1, and V(1,3) for Voice 2.

Line 25Ø increments Z by 1.

Line 26Ø creates a loop for the play section of the program by transferring control back to Line 23Ø.

The final section of the program is the mastermind behind manipulation of the DATA stored in the array. Whenever the computer encounters the flag in the DATA during the play portion of the program, it will transfer control down to the DATA control section (Lines 3ØØ-38Ø). Let's take a closer look at this section.

Line 3ØØ initializes the counter C. This counter keeps track of how many times the computer has played through the information stored in the array. The computer uses the flag to tell it when it has reached the end of the DATA.

Line 31Ø is an ON-GOTO statement linked up with the counter C. When C = 1, control is transferred to Line 32Ø. When C = 2, control passes to 34Ø, etc.

Line 32Ø reinitializes the array element counter Z to 3, thus starting the scale at the third note.

Line 33Ø sends the computer back to the play portion of the program with Z set at 3.

Line 34Ø sets Z to 5, thus starting the scale at the fifth note.

Line 35Ø sends the computer back to the play portion of the program with Z set at 5.

Line 36Ø sets Z to 1 for another scale starting at the first note.

Line 37Ø sends the computer back to the play portion of the program with Z set at 1.

Line 38Ø ENDs the program.

You can insert the following lines to see how the DATA CONTROL section interacts with the PLAY section of the program.

```
235 PRINT Z; V(Z,1);V(Z,2);V
(Z,3)

305 PRINT "DATA CONTROL"
```

Now when you RUN the program, the computer PRINTs out the values stored in the array and then the values being taken out of the array. Then it will play the music, using those values. The line "DATA CONTROL" printed on the screen means that the computer has just entered the third part of the program. Pay particular attention to the first column of numbers, representing the array element counter Z. They will clue you in to which pass of the loop the computer is making. The music is slow because the computer must PRINT all that information before each note.

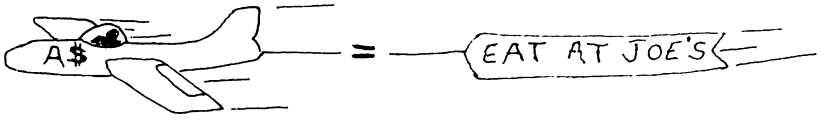
Entire books can be and have been written about arrays; this chapter is only an introduction. We suggest that you take some time to investigate their other possibilities. You will find that the time spent learning about them will greatly facilitate your ability to use complex DATA structures in your programs.

We'll take a final look at arrays in Chapter 20 when we'll use them to build a program that will turn your computer keyboard into a touch-sensitive chord organ.

18 String Functions

TI BASIC has many built-in functions for handling string variables. Strings, you'll remember, are groups of characters enclosed in quotation marks such as "Hello there!" or "A = B + C." The last character in a string variable name is always the dollar sign; for example, we could call "Hello there!" A$, or even HELLO$. Strings can be up to 255 characters long. You can link strings together with the ampersand (&). If A$ = "MOTOR" and B$ = "CYCLE," then A$ & B$ = MOTORCYCLE.

TI BASIC string functions work two ways: some functions evaluate a string to produce a number, and others evaluate a numeric expression to produce a string. The most common error you'll get when using strings and string functions is a STRING-NUMBER MISMATCH. This happens if you try to use a string, string variable, or string function to create a numeric result, or use numeric arguments, expressions, or variables where a string is required. For example, string functions that themselves end with a dollar sign, such as the STRing-number (STR$) function, always produce strings and cannot be used as part of a numeric expression.



ASCII VALUE—ASC (String or String Variable)

The ASCII value string function generates the ASCII-based character code for the first character of a string. ASC("S") is 83. ASC("SOAP") is also 83. If B$ = "SOAP" then ASC(B$) = 83. Keep in mind that

strings such as "MY NAME IS ALICE," which are sometimes called **string constants,** require quotation marks to be recognized by the computer as strings while string variable names with the "$" do not. (To review the ASCII character codes for all the characters on the keyboard, turn back to the appendix of the *User's Reference Guide* or see the TI reference card.)

The following program uses ASC to tell you what the ASCII code value is for any key you press. The actual ASC string function is lurking at the end of Line 12Ø behind the semicolon—ASC(A$) for the input of A$ in Line 11Ø.

```
100 CALL CLEAR

110 INPUT "ASC(X) FOR WHICH
KEY? ":A$

120 PRINT:"ASC(""";A$;""")=
";ASC(A$)::

130 GOTO 110
```

The empty space has an ASCII code just as any other character does. In order to see it in this program, however, you must bracket your empty space with quotation marks. If you were simply to hit the space bar and enter it as an input, the computer would have no way of telling if you meant to enter anything at all. As a result, A$ in ASC(A$) would be a string of zero length (null string), and the error message BAD ARGUMENT IN 12Ø would result.

CHARACTER—CHR$ (Numeric Expression)

The CHR$ string function is the logical inverse of the ASC function. CHR$ delivers the character corresponding to the value of the numeric expression enclosed in parentheses behind it. This value can be any number from 0 through 32767, but in practice, it will be one of the ASCII character codes between 32 and 159.

ASCII codes from 128 to 159 will return user-defined graphic characters (see Chapter 16). If you have redefined one of the normal keyboard symbols by renaming the ASCII value associated with it in a CALL CHAR statement, the CHR$ function will display your redefined character. This effect is demonstrated in the following program. Line 110 redefines the ASCII value of 46, normally the period, into a solid black block. If you

enter the value 46 into the program, you will see a block instead of a period. When you press FCTN:CLEAR, the block disappears and you get the period once again.

```
100 CALL CLEAR
110 CALL CHAR(46,"FFFFFFFFFFFFFFFF")
120 INPUT "CHR$(X) FOR WHICH
 VALUE?    (0 TO 32767):
" : A
130 PRINT:"CHR$(";A;")=";CH
R$(A)::
140 GOTO 120
```

The first two "CHR$"s are in quotation marks as part of the display when the program runs. The real function CHR$, which produces the character displayed after the equals sign, comes at the end of Line 130.

Using CHR$ to display user-defined characters can come in very handy for graphics programming. The following program uses CHR$ in two PRINT statements to produce a checkerboard pattern on the screen. The first PRINT statement, at Line 140, prints the character whose ASCII value is 33 (defined in Line 110) and then an empty space. The empty space is printed before, rather than after, the character displayed by CHR$(33) in Line 170. This pair of alternating print statements is repeated twelve times to fill the screen.

```
100 CALL CLEAR
110 CALL CHAR(33,"FEFEFEFEFE
FEFEFE")
120 FOR T=1 TO 12
130 FOR X=1 TO 14
140 PRINT CHR$(33);" ";
150 NEXT X
160 FOR Y=1 TO 14
170 PRINT " ";CHR$(33);
180 NEXT Y
190 NEXT T
200 GOTO 200
```

VALUE—VAL (String Expression)

If you have numbers in a string and you wish to use them in calculations, you need the VALue string function: VAL(A$) delivers the numeric value of A$. Be sure that the string whose VALue you asked for really has a valid numeric equivalent; otherwise the program you are running will stop, and you'll get a BAD ARGUMENT error message. For instance, VAL(B$) makes no sense if B$ is the word "BOOK."

You cannot perform numeric operations within the parentheses of a VAL string function, even if your strings have numerical equivalents. You may, however, combine the numerals of two or more strings by using the ampersand. For example, VAL(A$&B$) will produce 3310 if A$ is 33 and B$ is 10. This combining, or **concatenation**, of strings is not at all the same as adding numbers.

If the number you produce by chaining two strings together in a VAL function has more than 10 digits, the computer displays it on the screen in scientific notation—just as it would any other numeric value of that length.

When using VAL, avoid strings with no characters (empty strings). The string E$, where E$ = "" will stop a program containing VAL(E$). At the other extreme, a string longer than 254 characters in a VAL function will also stop a program.

The following program demonstrates some examples of the VAL function; it ends by using VAL on a string with no character. Line 180 shows that you can perform numeric operations involving two VAL functions. In this case VAL(A$), which is 12, is multiplied by VAL(B$), which is 3, to give 36.

```
100 A$="12"
110 B$="3"
120 C$=A$&B$
130 D$="-.0045"
140 E$="222222222222"
150 F$=""
160 PRINT VAL(A$)
170 PRINT VAL(A$&B$)
180 PRINT VAL(A$)*VAL(B$)
190 PRINT VAL(A$&B$&C$)
200 PRINT VAL(D$)
210 PRINT VAL(E$)
220 PRINT VAL(F$)
```

STRING NUMBER—STR$ (Numeric Expression)

The STRing number function is the logical inverse of the VALue function: it changes the value of a numeric expression into a string. STR$ is useful when you need to use numbers in a program as both numbers and strings. Changing numbers into strings lets you chain them together with the ampersand. The next program demonstrates this effect on Line 15Ø.

```
100 CALL CLEAR
110 INPUT "HOW OLD ARE YOU?"
:A
120 INPUT "WHAT YEAR IS THIS
?":Y
130 A$="YOU WERE BORN IN "
140 B$=STR$(Y-A)
150 PRINT A$&B$
160 GOTO 110
```

As Line 14Ø demonstrates, numeric operations *can* take place within the parentheses of a STR$ function. The resulting value is then converted into a string. If the number resulting has more than 10 digits, the resulting string will take the form of a number with scientific notation—despite the fact that it is now a string. To produce a string number which is more than 10 digits, without scientific notation, you must link two strings together with an ampersand. The following examples illustrate these effects.

```
PRINT STR$(22222222222+22222
222222)
4.44444E+10
```

```
PRINT STR$(22222222222*(STR$
(33333333333)
2.22222E+103.33333E+10
```

```
PRINT STR$(22222222)&STR$(33
333333)
```

In the third example above we again see scientific notation because each time STR$ is performed, the number is longer than ten digits.

Once the STR$ function has been performed on a number, that number can only be manipulated as a string. To perform numerical operations once more, you must convert the string number created by the STR$ function into a number (again), by using the VALue string function (*).

LENGTH—LEN (String Expression)

The LEN string function tells you how many characters there are in a string or string expression. You may place either a string name within the parentheses or a string itself. LEN(X$) gives the number of characters in a string named X$. You may also combine two or more strings with ampersands within the parentheses; and the value returned will be the total number of characters of all the combined strings. You must bracket a string, in quotation marks, within the parentheses. LEN ("DOGFOOD"), for example, is 7.

If you use a string with no characters, (C$ = ""), with LEN; you'll get a LENgth of zero. But you cannot perform numeric operations within the parentheses. If you wish to use numeric operators you must place them *between,* rather than within, LEN functions.

```
100 A$="TI BASIC IS FUN"
110 B$="TO USE"
120 C$=""
130 PRINT LEN(A$)
140 PRINT LEN(B$)
150 PRINT LEN(A$&B$)
160 PRINT LEN(A$)*LEN(B$)
170 PRINT LEN(C$)
180 PRINT LEN("EDUCATION")
```

POSITION—POS (A$, B$, Numeric Expression)

The POSition function finds the starting position of one string within another string. The numeric expression included within the parentheses at the end tells the computer to start the search for the position of the second string at a certain location within the first string. If, for instance, the numeric expression yields a value of 4, the search for the second string will begin at the fourth character of the first string. When the value derived from the numeric expression is applied in the POS function, it will be rounded to the nearest whole number, if necessary.

If the value of the numeric expression is rounded down to a zero, a BAD VALUE error message results, and the program stops. If the value of the numeric expression is larger than the number of characters in the first string, a value of zero is returned for the position of the second string: the computer returns a zero when it cannot find the second string within the first string.

POS(A$,B$,14) tells the computer to look for the number of the character in the A$ string at which the B$ string begins, and to begin

this search procedure at the 14th character in A$. In order for the position of the second string to be found, it must occur after the point at which the computer has been instructed to begin its search procedure in the first string. In the following program

```
100 D$="45555"

110 E$="4"

120 PRINT POS(D$,E$,1)

130 PRINT POS(D$,E$,2)
```

the computer returns the number 1 in Line 120 and the number 0 in Line 130. This is because E$, or "4," cannot be found if the search process begins at the second character of D$.

The next program uses a FOR-NEXT loop to search for the position of B$ within A$ at every character of A$. FOR X=1 TO 20 in Line 120 sets the starting point of the search for B$ at the first, second, third character of A$ until the 20th character has been reached. A$ actually has only 19 characters, but a 20th value for the search procedure shows that it is possible (if useless) to search beyond the length of the first string.

When you RUN the program, you'll see two columns of numbers. The numbers within parentheses represent the starting point of the search within A$. The number paired with each of these starting locations is the position of the first character of B$ ("CA") within A$. Notice that this does not mean that just the "C" of B$ is being sought. The computer will not find the position of B$ unless all of B$ occurs at some point within A$. In this program, for instance, the "CA" of B$ is not found after the first character of A$, where it appears in the word CAN, until it appears again in the word CATS which begins at the tenth character of A$. The computer does not recognize the "CO" in the world COLD which falls between these two points, as CO is not CA.

```
100 A$="CAN COLD CATS CATCH"

110 B$="CA"

120 FOR X=1 TO 20

130 PRINT "(";X;")";POS(A$,B
$,X),

140 NEXT X
```

STRING SEGMENT—SEG$ (String Expression, Numeric Expression #1, Numeric Expression #2)

If you wish to define a segment of a string to use later in a program, use the string SEGment function. SEG$ returns a substring of your string, derived from the string expression placed within the parentheses. This string expression is followed by two numeric expressions. The value of the first one determines which character in the first string will be the first character of the new substring. If this value were 12, for instance, then the 12th character of the original string would become the first character of the new string segment. The number of characters in the new string segment is determined by the value of the second numeric expression. In the following example

```
100 A$="STRING SEGMENT FUNCT
ION"
110 PRINT SEG$ (A$,8,7)
```

the word SEGMENT is produced because the first character of the new substring begins at the eighth character of A$ (or "S"). The number 7 defines the length of the new string segment as seven characters, thus comprising the seven letters in the word SEGMENT.

If the number specifying the start of the string segment is bigger than the total number of characters in the original string, SEG$ comes up with a string without any characters (null string). You'll also get a null string if the number specifying the length of the new string segment is zero. If this number is larger than the number of characters remaining in the orignial string after the start of the new segment, you get those remaining characters. If Line 110 in the last example read

```
110 PRINT SEG$ (A$,8,50)
```

you'd get SEGMENT FUNCTION. Although you have asked for a string segment with 50 characters, the computer has nothing left to read after the "N" of function.

Two kinds of values in the numeric expression of the SEG$ function will stop a program. If you give a new segment a starting point less than or equal to zero, you'll get a BAD VALUE error message. If you specify segment length as less than zero, the program also stops. The computer rounds off decimal fractions in either of the numeric expressions to the nearest whole number. A value of −.49 in string segment length, for instance, would be rounded off to zero and give a null string instead of stopping the program.

The next program is really two little programs in one. Lines 11Ø to 15Ø perform various SEG$ functions on the A$ string defined in Line 1ØØ. After this, a CALL KEY statement on Line 16Ø lets you look at the results of the first half of the program before setting off the FOR-NEXT loop in Lines 180 to 200 by pressing any key.

The first series of SEG$ functions are straightforward manipulations of the A$ string, such as we've already seen. The FOR-NEXT loop, however, produces a rather impressive, but perhaps confusing, effect. Thirty-four new segments of the A$ string are printed, for the values of X from 1 to 34, where X is used to determine both the starting location of each segment as well as its length. The resulting pattern printed on the left half of the screen, is a large wedge pointing to the right. When X = 1, the first character in A$ is printed in a segment one character in length. When X = 2, the second and third characters of A$ are printed, because this segment starts at the second character and is two characters long. The FOR-NEXT loop continues in this fashion until the 34th character of A$ becomes the first character of a segment.

The segments begin to get shorter when X = 18 because A$ has run out of new characters to add. Hence, even though longer segments are specified in the SEG$ functions from X = 18 onward, the lines grow shorter as each segment begins at a later point in the A$ string.

If you redefine the characters within strings with CALL CHAR statements and then call up your redefined characters in SEG$ function loops like this one, you can create some really interesting graphics.

Since new segments called in the FOR-NEXT loop begin at successive locations in the collection of characters which make up A$, you will be able to read A$ in the leftmost vertical column on the screen.

```
1ØØ A$="I NEVER MET A STRING
 I DIDN'T LIKE"
11Ø PRINT SEG$(A$,1,11)
12Ø PRINT SEG$(A$,13,1)
13Ø PRINT SEG$(A$,15,6)
14Ø PRINT SEG$(A$,22,8)
15Ø PRINT SEG$(A$,31,4)
16Ø CALL KEY(3,R,S)
17Ø IF S=Ø THEN 16Ø
18Ø FOR X=1 TO 34
19Ø PRINT SEG$(A$,X,X);" ";X
2ØØ NEXT X
```

19 More Graphics

In Chapter 15 we introduced the various graphics related statements in TI BASIC as well as a few elementary concepts in graphics programming. Here we will look at some techniques which can increase the power and sophistication of your program's graphics.

FOR-NEXT LOOPING

No matter what the program, but especially for complex graphics, it is best to make your instructions as compact as possible. You could tell the computer to do one thing per program line, but you'd end up with long programs and tedious typing and debugging sessions; you might even run out of memory. FOR-NEXT loops can help cut down the number of lines you need.

Let's say, for example, that you're writing a program to draw 24 lines across the screen, so it looks like a sheet of notebook paper. You could enter:

```
100 CALL CLEAR
110 CALL CHAR(36,"0000000FF00
000000")
120 FOR X=1 TO 24
130 CALL HCHAR(X,1,36,32)
140 NEXT X
150 GOTO 150
```

Here, the FOR-NEXT loop in Lines 120-140 performs the HCHAR statement 24 times for each of the row values from 1 to 24. You could do the same thing with a separate HCHAR statement for each row, but the program would be 27 lines long, not six!

We can use any variable in a graphics statement in a FOR-NEXT loop. If you name the variable in your FOR-NEXT statement (the X in Line 13Ø, for example), you can also use it in a STEP clause. For instance, FOR X = 1 TO 24 STEP 2, would draw the lines on every other row. You can also manipulate the variables in your graphics statements with a loop without naming them in the FOR-NEXT statement, thus letting you have more than one variable at work within the loop:

```
100 CALL CLEAR
110 CALL CHAR(36,0000000FF00
000000")
120 Y=1
130 FOR X=1 TO 24
140 CALL HCHAR(X,Y,36,8)
150 Y=Y+1
160 NEXT X
170 GOTO 170
```

Here we are manipulating both the row and column locations of the HCHAR statement with the X and Y variables, respectively. You must create an initial value for the variable Y before the FOR-NEXT loop is performed. In this case the value of Y is already established when the loop begins. Each time through the loop, the value of Y is increased by one on Line 15Ø. Thus each new line appears one column farther to the right on the screen.

Sometimes you may need an IF-THEN statement to prevent bad values in your graphics statments. Adding the following lines

```
150 Y=Y+2
155 IF Y=33 THEN 157 ELSE 16
0
157 Y=10
```

increases the column value in the HCHAR statement by two each time through the loop, giving Y = 1, 3, 5, 7, 9, 11, etc. The 17th time through the loop, the column value is 33, which could cause a BAD VALUE error message at Line 14Ø because there is no column 33 on the screen. Lines 155 and 157 reset the value of Y when it is too high.

CALLING COLOR

Because CALL COLOR only CALLs a foreground and a background color for a set of eight characters, you must plan carefully for programs

where you want many colors to appear simultaneously.

In the next program, the screen fills from left to right with two sets of 16 vertical strips. Each vertical strip is one of the 16 possible colors generated by the computer. To get this effect, characters from each of the 16 character sets must appear in the VCHAR statement.

The program is set up as two FOR-NEXT loops. The first loop CALL for COLOR 16 times, once for each character set as well as for foreground and background colors. The second creates a character CH and displays it 32 times. The value of CH is initialized at Line 140, before the second FOR-NEXT loop begins. Within the second FOR-NEXT loop, however, CH (the character number) is changed at each pass by the adding eight to it in Line 180. This also has the effect of placing each displayed character in the next higher character set—and thus in a different color—already determined for that set in the initial CALL COLOR loop. If the value of CH gets too high, Lines 190 and 200 reset it at the original 33.

```
100 CALL CLEAR
110 FOR CO=1 TO 16
120 CALL COLOR(CO,CO,CO)
130 NEXT CO
140 CH=33
150 FOR Y=1 TO 32
160 CALL CHAR(CH,"FFFFFFFFFF
FFFFFF")
170 CALL VCHAR(1,Y,CH,24)
180 CH=CH+8
190 IF CH=161 THEN 200 ELSE
210
200 CH=33
210 NEXT Y
220 GOTO 220
```

You may not see all 32 columns because many televisions drop off a column or two on either the far left or right hand side of the screen. The light green column between the white and black columns in the center of the screen is actually transparent, so the screen colors show through.

When you press FCTN:CLEAR you'll see a different character in each column: as the value of CH changes in the FOR-NEXT loop, it takes on the character code values normally belonging to those characters. The solid block of four black columns are user-defined character codes between 128 and 159. Because no other character definitions exist within this range, the character displayed while the program was running is still present.

At this point, we can only see one color in each column for two reasons. First, the character defined in Line 160 is a solid block without any background. Second, the loop in Lines 110-130 have made the foreground and background the same value. To get different foreground and background colors, let's redefine the character CH in Line 160 and add the following lines:

```
160 CALL CHAR (CH,"ØFØFØFØFFØ
FØFØFØ")
112 RANDOMIZE
114 CF=INT(16*RND)+1
116 CB=INT(16*RND)+1
120 CALL COLOR(CO,CF,CB)
```

Now we have assigned a random value from 1 to 16 to the foreground color, called CF, and given a similar value to the background CB. A different random value is chosen for both CF and CB for each of the sixteen character sets. Our redefined character CH is now made up of four smaller blocks, two for foreground color and two for background color.

Now let's add one final twist to this program. Try adding these two lines:

```
135 IF D<Ø THEN 11Ø
215 D=-1
220 GOTO 11Ø
```

After the computer RUNs through the entire program once, the CALL COLOR loop is performed again and again. This is because the computer automatically assigns a value of zero to any variable not otherwise defined in a program. The first time the program encounters Line 135, D therefore equals zero; the IF-THEN condition does not hold and the program continues Once D = −1 in Line 215, however, the program returns to Line 11Ø, restarting the CALL COLOR loop.

Using CALL COLOR can be difficult if you wish to use many colors and display text simultaneously. Because you must use different character sets to get many colors simultaneously, you will probably need to use characters within the character sets populated by either the alphabet or numbers. You could stop using the lower-case letters or (although this takes considerable effort) you could relocate the characters you wish to leave out of your color manipulations, treating them as redefined characters in one or several character sets. But if you do you'll also have to write in the appropriate CALL KEY logic to find your relocated characters when you press keys on the keyboard, and that can be really

tedious.

A better solution is to write your program to display graphics and text on the screen at different times. With CALL CLEAR and FOR-NEXT loops to redefine the colors for a series of character sets you can make reasonably quick transitions from graphics to text. (Unfortunately, keeping discrete sections of the screen reserved for graphics and text is difficult because PRINT and INPUT displace graphics content by moving the entire screen up from the bottom.)

You can print messages at selected locations on the screen, provided they're stored in the program as strings, without disturbing graphic contents elsewhere by using CALL HCHAR or CALL VCHAR in conjunction with string functions. This technique will be explained later in this chapter.

STRINGS AND STRING FUNCTIONS

You can use strings and string functions to produce graphics if you create strings whose characters you have redefined as graphics characters. The trick is to invent strings that are spatial representations of the graphics you desire and then manipulate the *strings*. If you don't mind designing a screen from the bottom up, you can easily use PRINT statements. But if graphics are already on the screen, and you wish to add new text or graphics without disturbing them, you'll have to display your new material with an HCHAR or a VCHAR statement combined with various string functions. You might want to review the sections in Chapter 18 on the character (CHR$) and the segment (SEG$) string functions before going on.

Using PRINT

When you are constructing a graphics screen by PRINTing strings, you can control the location of your graphics by putting spaces within the PRINT statements. You can either include the spaces within quotation marks in the PRINT statements, or use PRINT TAB (X), where X is the desired number of spaces from the left.

The following program demonstrates these effects. Notice that the A, B, and C$ strings are concatenated with ampersands on Line 200.

```
100 A$="0"
110 B$="1"
120 C$="2"
130 CALL CHAR(48,"F0F0F0F00F
0F0F0F")
140 CALL CHAR(49,"CCCCCCCCC
```

```
CCCCCC")
150 CALL CHAR(50,"0000FFFF00
00FFFF")
160 FOR X=1 TO 28
170 PRINT A$;B$;" ";C$;" ";
180 NEXT X
190 PRINT ::::
200 PRINT TAB(13);A$&B$&C$
210 PRINT ::::
220 GOTO 160
```

Now let's try using the STRing number function (STR$) for graphics. Remember that STR$ takes the numeric expression X and makes it into a string in the expression STR$(X). Numbers can be easily manipulated in programs with simple arithmetic operators. The problem with numbers as print display items, however, is that they leave leading and trailing spaces that you can't get rid of. To take advantage of numbers without the side effect of these spaces, we can use STR$ to change a number into a string before printing it. We can then concatenate any strings we have formed.

The next program demonstrates using STR$ in this way. We first set up the variable A = 10. We then redefine both 1 and 0 as the same character. Each time the FOR-NEXT loop is performed, A is multiplied by 10, and STR$(A) becomes one character longer. If you change Line 140 to FOR X = 1 TO 10 you will get scientific notation in your display. To avoid this, you must concatenate two or more strings each 10 digits or fewer in length to create one larger image.

```
100 CALL CLEAR
110 A=10
120 CALL CHAR(48,"F0F0F0F00F
0F0F0F")
130 CALL CHAR(49,"F0F0F0F00F
0F0F0F")
140 FOR X=1 TO 8
150 PRINT STR$(A)&STR$(A)
160 A=A*10
170 NEXT X
180 PRINT ::
190 GOTO 110
```

Using SEGment and LENgth String Functions

The SEGment and LENgth string functions give you some potent graphics techniques. SEG$ is followed by your string's name as well as two numeric expressions identifying the start and length of a new string segment. SEG$ lets you have a master string of various characters from which you can select any segment for display.

The next program shows off this feature. We have a FOR-NEXT loop in Lines 13Ø to 15Ø nested within another that begins at Line 12Ø and ends at Line 17Ø. The inner loop merely PRINTs a character in A$, as defined by X, ten times. The outer loop tells the computer to fetch the next character—or value for X—in the A$ string, after each set of ten characters has been printed. This is done until the value of X = LEN(A$), or the total number of characters in A$. If you had defined a new character for each letter in A$, you could then create different graphics images.

```
100 CALL CLEAR
110 A$="ABCDEFGHI"
120 FOR X=1 TO LEN(A$)
130 FOR Y=1 TO 10
140 PRINT SEG$(A$,X,1);
150 NEXT Y
160 PRINT ::
170 NEXT X
```

Printing Text With HCHAR

So how do we display messages on the screen without disturbing graphics already there? By displaying a string with an HCHAR or VCHAR character repetition statement. First, we have to supply an ASCII character code for each string character we wish to show. We can do this by using the ASCII string function with SEG$. For instance, if the string A$ = "YELLOW" then

```
PRINT ASC(SEG$(A$,1,1))
```

will display the number 89, the character code for an upper-case Y. By picking out single character string segments we can thus supply an HCHAR statement with all the ASCII values it needs to display the string for us anywhere we want on the screen.

We must also determine the column and row locations for our displayed characters. For this we can use IF-THEN statements to manipulate the row and column variables and to prevent bad values.

The next program demonstrates these techniques. The INPUT prompts let you enter your message and select row and column locations. Be sure to put a comma between the numbers used for the row and column location. Later, you might try rewriting Line 15Ø to read:

```
150 FOR C=LEN(A$) TO 1 STEP
-1
```

and the program will print your message backward!

```
100 CALL CLEAR
110 INPUT "PRINT:":A$
120 INPUT "ROW?,COLUMN?:":X,
Y
130 CALL CLEAR
140 S=0
150 FOR C=1 TO LEN(A$)
160 CHAR=ASC(SEG$(A$,C,1))
170 CALL HCHAR(X,Y+S,CHAR)
180 S=S+1
190 IF Y+S=30 THEN 200 ELSE
230
200 S=0
210 Y=3
220 X=X+1
230 NEXT C
240 GOTO 110
```

USING ARRAYS IN GRAPHICS

Because the data stored in arrays need not be read more than once into a program, arrays are especially useful for setting up complex displays and repeated graphics effects.

The following program uses arrays and draws either the letter A, B, or C on the screen. Each of these letters is seven character spaces high and five spaces wide. Line 110 DIMensions a string array called LE$ in two dimensions, 7 and 3. This gives an array containing three groups of seven data items each. You can follow this pattern in the three DATA statements at the end of the program: each data item contains five character, each item is separated from the next by a comma, and each DATA line has seven data items. The three DATA lines are the three groups in the array's second dimension.

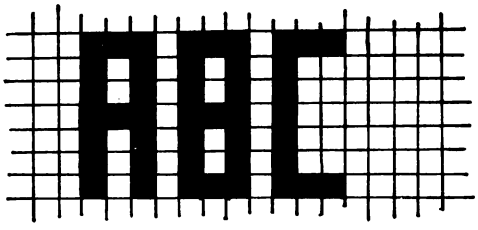
The beauty of this arrangement is that every time we ask for LE$(4,2), for instance, the computer reads in the fourth data item in the second group. In this program, that corresponds to the five characters that form the center line of our large letter B.

Using a READ statement after DIMensioning the array, we ask the computer to store it. This it does with two FOR-NEXT loops: the loop associated with the array's first dimension is nested within that associated with the second dimension. Once this process is complete, the computer has mapped out the data structure which makes up the large letters and has it "on tap."

Starting at Line 310 is another set of nested FOR-NEXT loops. Much as the first nested loops READ the data into the array, the second loop puts out the data, displaying it on the screen. The programming tech-

nique here, on Line 330, is the same as the one we used in the "Printing Text With HCHAR" section of this chapter. The only difference is that now that we are dealing with LE$ as a two-dimensional array. So we must always refer to the array as LE$(X,Y) where X and Y are the variables associated with each dimension.

```
100 CALL CLEAR
110 DIM LE$(7,3)
120 FOR Y=1 TO 3
130 FOR X=1 TO 7
140 READ LE$(X,Y)
150 NEXT X
160 NEXT Y
170 CALL CHAR(81,"FFFFFFFFFF
FFFFFF")
180 CALL CHAR(46,"0000000000
000000")
190 INPUT "ROW?(1 TO 18):":R
OW
200 IF (ROW<1)+(ROW>18)THEN
190
210 INPUT "COLUMN?(1 TO 28):
":COL
220 IF (COL<1)+(COL>28)THEN
210
230 PRINT "A,B, OR C?:"
240 CALL SOUND(180,1400,5)
250 CALL KEY(3,R,S)
260 IF S=0 THEN 250
270 IF (R<65)+(R>67) THEN 25
0
280 CALL CLEAR
290 R=R-64
300 FOR X=1 TO 7
310 FOR CH=1 TO 5
320 CHAR=ASC(SEG$(LE$(X,R),C
H,1))
330 CALL HCHAR(ROW,COL+SP,CH
AR)
340 SP=SP+1
350 NEXT CH
360 SP=0
370 ROW=ROW+1
380 NEXT X
390 GOTO 190
400 DATA .QQQ.,Q...Q,Q...Q,Q
QQQQ,Q...Q,Q...Q,Q...Q
410 DATA QQQQ.,.Q..Q,.Q..Q,.
QQQ.,.Q..Q,.Q..Q,QQQQ.
420 DATA .QQQ.,Q...Q,Q....,Q
```



Line 11Ø sets up a two-dimensional array LE$(X,Y).

Lines 12Ø-16Ø READ the array into memory, using two FOR/NEXT loops built around a READ statement in Line 14Ø.

Lines 17Ø-18Ø redefine the characters for upper case Q and the period to a solid block and empty space.

Lines 19Ø-29Ø allow you to decide which letter to display and where. Lines 19Ø and 21Ø define the INPUT variables for ROW and COL location. On Line 25Ø a CALL KEY statement lets you choose which letter is displayed. The return variable values in the CALL KEY statement for the keys of A, B, and C are 65, 66, and 67 respectively. We need only subtract 64 from these values in Line 29Ø to get 1, 2, or 3 for "R" in Line 32Ø.

Lines 3ØØ-38Ø contain a double FOR-NEXT loop that puts the chosen letter on the screen. Line 3ØØ starts the FOR-NEXT loop for each of the seven layers of five characters making up each letter. Line 31Ø starts the FOR-NEXT loop that displays each of the five characters in each piece of data, which form one of the seven layers of the letter to be displayed. Line 32Ø generates the ASCII character code needed by the HCHAR statement in the next line to generate the large letter, character by character. The variable CHAR is the code number produced by the ASC string function and represents the value of a particular one-character segment of LE$(X,R),CH,1. Here X is one of the seven five-character layers of our large letter. R is the value 1, 2, or 3, stipulating which element of the second dimension of LE$(X,R) has been input by the user on choosing to display an A, B, or C. CH is the CHaracter at which the particular segment of LE$(X,R) is to begin, from 1 to 5. The 1 at the end indicates that each segment beginning at CHaracter CH is only one character long. Line 33Ø is a CALL HCHAR statement that displays the character with the ASCII value of CHAR at the ROW and COLumn locations you entered earlier. These row and column locations are manipulated by adding the value of the

variable SP to the column location each time the loop is performed. To begin with, SP is an uninitialized variable whose value is zero. This value is restated each time a new layer of a letter is begun by setting SP to zero in Line 36Ø.

Line 34Ø moves the column location one space to the right.

Line 35Ø sends the CH loop back for another character.

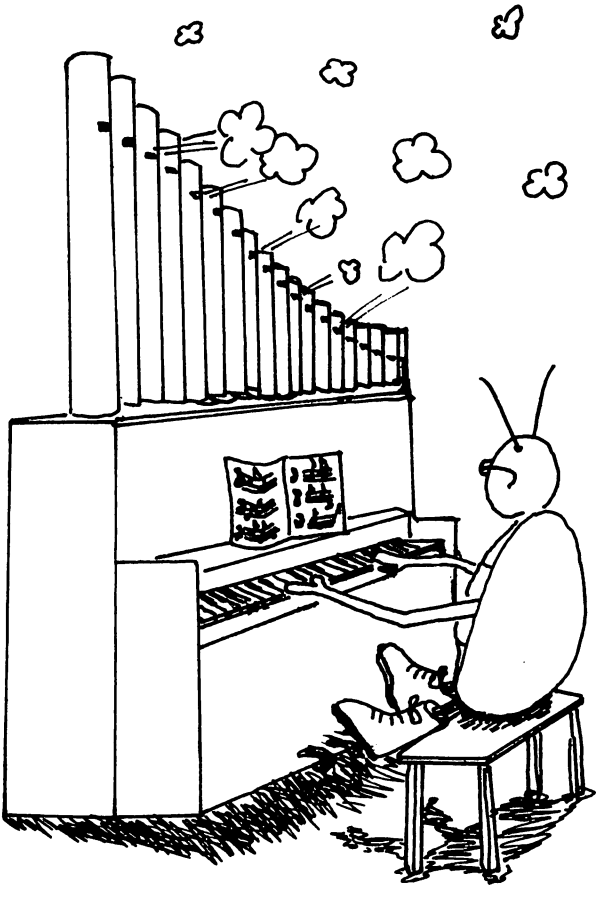
Lines 36Ø-37Ø reset the column position for the next layer of the letter being constructed. The ROW location is also moved down one to set the new layer.

Line 38Ø sends the X loop back to draw another five-character section of the letter.

Line 39Ø sends the program back to ask for new INPUT for a new letter on Line 19Ø.

Lines 4ØØ-42Ø are the DATA that go into array LE$(X,Y). These lines contain the raw materials for our large letters, A, B, and C.

20 Live Time on the Keyboard



It's time now to face the music—an 80-line program. Long? Yup, but then, so is most developed software. And this software makes music. The program includes arrays, CALL KEY, the sound chip, print formatting, and a whole series of graphics commands. You've used them all before; now let's combine them in a full-scale production.

Just so you know what you're getting into, here's what happens. First, the screen clears and you'll see a short set of instructions. Then the screen goes black and a white musical staff appears, line by line. After the staff has been drawn, musical notes in different colors will appear on the staff, and large block print, consisting of upper-case O's combined to make letters, will announce COLOR above the staff and ORGAN below the staff. When the graphics are all on the screen, you can start making music with your computer. The middle two rows of the keyboard correspond to the black and white keys of a piano: WER YU OP/ are the black keys, and ASDFGHJKL: are the white keys. Each time you press one of these keys, a pre-programmed three-note chord sounds for as long as you hold down the key. When you take your hand off the key, the chord stops, and the notes on the staff change colors.

Here now is the program in all its lengthy glory. Type it in carefully, RUN it, and check for bugs. If you find any (and in a program this long you probably will), work them out until the program RUNs properly. Then we'll look at the program in detail.

```
100 REM *** COLOR ORGAN ***
110 CALL CLEAR
120 PRINT TAB(9);"TI CHORD O
RGAN"::::::
130 PRINT "      PROGRAM DES
IGN BY"::"    MIDNIGHT MADNE
SS,  INC." ::::::::
140 PRINT " WE'RE HERE....YO
U'RE THERE"::
150 GOSUB 590
160 CALL CLEAR
170 PRINT "   PRESS ONE OF T
HESE KEYS"::::::::
180 PRINT " WER YU
OP/"
190 PRINT "ASDFGHJ
KL:"::
200 PRINT "(MIDDLE TWO
ROWS)":::::
210 PRINT " TO PLAY A THREE
NOTE CHORD" :::::
220 FOR T=1 TO 2000
```

```
230 NEXT T
240 CALL CLEAR
250 CALL SCREEN(2)
260 PRINT "    000 000 0   0
00 000"
270 PRINT "    0   0 0 0   0
 0 0 0"
280 PRINT "    0   0 0 0   0
 0 00"
290 PRINT "    000 000 000 0
00 0 0":::::::::::::
300 PRINT "    000 000 000  0
00 0 0"
310 PRINT "    0 0 0 0 0    0
 0 00 0"
320 PRINT "    0 0 00  0  0 0
00 0 00"
330 PRINT "    000 0 0 0000 0
 0 0  0":::
340 CALL COLOR(13,16,1)
350 FOR Y=10 TO 14
360 CALL CHAR(128,"0000000000
00000FF")
370 CALL HCHAR(Y,3,128,28)
380 NEXT Y
390 CH=40
400 FOR W=5 TO 28 STEP 1.66
410 RANDOMIZE
420 Q=INT(13*RND)+3
430 R=INT(6*RND)+10
440 SET=(CH-24)/8
450 CALL COLOR(SET,Q,1)
460 CALL CHAR(CH,"3C7EFFFFFF
FF7E3C")
470 CALL CHAR(129,"0001010101
0101FF")
480 CALL HCHAR(R,W,CH)
490 IF R=10 THEN 500 ELSE 53
0
500 CALL CHAR(130,"0001010101
0101011")
510 CALL HCHAR(R-1,W,130)
520 GOTO 540
530 CALL HCHAR(R-1,W,129)
540 CH=CH+8
550 IF CH=120 THEN 560 ELSE
570
560 CH=40
570 NEXT W
580 GOTO 760
```

```
590 DIM A(18)
600 DIM B(18)
610 DIM C(18)
620 DIM D(18)
630 FOR N=1 TO 18
640 READ A(N)
650 NEXT N
660 FOR N=1 TO 18
670 READ B(N)
680 NEXT N
690 FOR N=1 TO 18
700 READ C(N)
710 NEXT N
720 FOR N=1 TO 18
730 READ D(N)
740 NEXT N
750 RETURN
760 CALL KEY(3,X,Y)
770 IF Y=0 THEN 760
780 IF (Y=1)+(Y=-1)THEN 790
790 I=1
800 IF X=A(I)THEN 840
810 I=I+1
820 IF I<=18 THEN 800 ELSE 8
90
830 IF X<>A(18)THEN 890
840 CALL SOUND(-4000,B(I),0,
C(I),0,D(I),0)
850 CALL KEY(3,X,Y)
860 IF Y=0 THEN 890
870 IF Y=1 THEN 790
880 IF Y=-1 THEN 840
890 CALL SOUND (-1,40000,30,4
0000,30,40000,30)
900 RANDOMIZE
910 Q=INT(13*RND)+3
920 S=INT(10*RND)+2
930 CALL COLOR(S,Q,1)
940 CALL COLOR(6,Q,1)
950 GOTO 760
960 DATA 65,83,68,70,71,72,7
4,75,76,59,87,69,82,89,85,79
,80,47
970 DATA 175,196,220,247,262
,294,330,349,392,440,185,208
,233,277,311,370,415,466
980 DATA 220,247,262,294,330
,349,392,440,494,523,233,247
,311,370,466,554,622
990 DATA 262,294,330,349,392
,440,494,523,587,659,277,311
,349,415,466,554,622,740
```

LINKS IN A CHAIN

This program can be broken down into individual modules, each a link in the chain of overall program logic.

Link 1

The first link is the title page, created by the series of PRINT statements that come after Line 1ØØ clears the screen. The title page comprises Lines 12Ø-14Ø. Normally, we would need a time loop to keep the display on the screen. But in this case, Line 15Ø sends the computer into a subroutine starting at Line 59Ø, tying it up and thus keeping the display on the screen. The subroutine is the second link.

Links 2 and 3

The subroutine begins at Line 59Ø and continues all the way to Line 75Ø, which contains the RETURN statement. This subroutine loads four arrays. While the computer loads the the arrays, the title page stays on the screen.

We named the four arrays A, B, C, and D, each DIMensioned in Lines 59Ø-62Ø to hold 19 elements. We in fact use only 18 elements; this is just another way to call the first element in our array 1, without using the OPTION BASE statement.

The DATA bank in Lines 96Ø-99Ø is the third link in the chain. The DATA is loaded into the arrays in Lines 63Ø-74Ø. Lines 63Ø-65Ø load array A using the DATA from Line 96Ø of the DATA bank. Array A holds the character codes for the letters forming the organ keyboard.

Lines 66Ø-68Ø load array B using the DATA from Line 97Ø of the DATA bank. Array B holds the frequency information for Voice 1. Eighteen different frequency numbers are loaded into this array.

Arrays C and D are loaded in the same way by Lines 69Ø-71Ø and 72Ø-74Ø, respectively. These arrays hold the frequency information for Voice 2 and Voice 3. Again, there are 18 notes for each voice.

The RETURN statement in Line 75Ø ends this section of the chain and transfers control to the next.

Link 4

The RETURN statement moves the computer to Line 16Ø, which clears the title page display. Control then passes to the next line, which begins a series of PRINT statements that tell you which keys to push. This display is held on the screen by a time loop in Lines 22Ø and 23Ø.

Link 5

The fifth link of our chain controls the permanent graphics display. The screen is cleared in Line 24Ø and then turned black in Line 25Ø. This sets the stage for the block printing.

The block print is created in Lines 26Ø-33Ø using PRINT statements with the letter O and print separators. The procedure is simple to understand, but very exacting.

CALLing the screen black makes it possible to print all these Os on the screen without your seeing it happen. This is because the letters, too, are black; you will not see the block letters until the CALL COLOR statement in Line 45Ø is executed.

After the block characters are printed, the computer moves on to its next set of graphics statements (Lines 34Ø-38Ø), which will create the musical staff.

Line 34Ø defines the staff color as white with a transparent background. The character set is 13.

Lines 35Ø-38Ø form a loop that places the staff on the screen. Line 35Ø defines the loop and generates values for the variable Y. Line 36Ø defines the character for part of a single line; Line 37Ø places this line on the screen at the a proper location. This location is determined by the value of Y for the row, 3 for the starting column, and 28 for the number of character repetitions. The loop is performed five times, drawing a five-line staff.

Staff complete, the computer moves on to its next task, the rather complex job of placing notes on the staff. This is accomplished by Line 39Ø and the long loop between Lines 4ØØ-57Ø:

Line 39Ø defines the variable CH = 4Ø. This variable will be used later in the loop as a value for a character.

Line 4ØØ initiates the loop for placing the note heads and stems on the staff.

Line 41Ø randomizes.

Line 42Ø creates a random color value Q for each pass through the loop. The number will fall between 3 and 15.

Line 43Ø creates a random value for row placement. The number will fall between 10 and 15 (determining on which line of the staff the note will appear).

Line 44Ø assigns a value for a character set to the variable SET.

Line 45Ø SET and Q will be redefined each time through the loop.

Line 46Ø defines the character for the note head (the round part).

Line 47Ø defines the character for the note stem for all the notes except those above the top line of the staff.

Line 48Ø displays the note head using a row location generated by Line 43Ø,

a column location generated by Line 4ØØ, and a character code initialized in Line 39Ø and incremented each time through the loop by Line 54Ø.

Line 49Ø tests IF the note head appears above the top line of the staff. IF so, THEN 5ØØ ELSE 53Ø.

Line 5ØØ defines a stem for a note that appears above the top line of the staff.

Line 51Ø displays the stem for a note head that appears above the staff one space above the note head.

Line 52Ø skips over 53Ø, which displays the normal stem.

Line 53Ø displays the stem for all notes below the top line of the staff—one space above the note head.

Line 54Ø increments the ASCII character code into the next character set.

Line 55Ø tests if CH = 12Ø. IF it does, THEN 56Ø ELSE 57Ø.

Line 56Ø resets CH to 40. This safeguards against altering the values in the character sets holding the note heads and staff characters.

Line 57Ø loops back for the next note.

This loop solves an interesting graphics problem. Because the note is so large, it intersects with the staff lines. We decided to let the bottom of the note replace part of the staff line, but create a full line above the note head. We accomplished this by using the two different stems, one for notes falling below the top line of the staff and one for notes falling above the top line of the staff. The first stem looks like this: (▟) while the second looks like this: (▌).

After the graphics have been completed in Line 57Ø, Line 58Ø tells the computer to leave this module and move on to the next (GOTO 76Ø).

Link 6

The next link in the program logic chain starts on Line 76Ø and ends on Line 95Ø. This section of the program converts the computer keyboard into an organ keyboard. It also changes the color of the graphics display whenever a key is released, and no new key is immediately pressed.

The first step is a CALL KEY statement in Line 76Ø. The return variable is called X and the status variable is called Y.

The next step is to evaluate the character returned by the CALL KEY; this happens in Lines 77Ø-83Ø. These lines also transfer control to other sections of the program according to the values received.

In Line 77Ø, if no key is pressed the computer returns to the CALL KEY statement in the previous line. In Line 78Ø, if a new key is pressed, or if the same key is pressed, the computer moves to the next line. Line 79Ø resets I to 1.

Lines 8ØØ-83Ø check if the character code of the key being pressed equals any of the 18 codes stored in array A. IF so, THEN control passes to the CALL SOUND statement in Line 84Ø which plays a chord. IF not, THEN control passes to a second CALL SOUND statement in Line 89Ø which plays a millisecond of silence. Thus you play chords when you hit the keys whose codes reside in array A and play silence when you hit keys outside the range of array A.

If you do press one of the chord keys, the CALL SOUND statement of Line 84Ø comes into play. The value of the array element counter (I) has been incremented in Line 81Ø for each pass of the search loop. I = 5 means that the character code of the key pressed equals the number stored as the fifth element of the A array. The CALL SOUND statement then uses the fifth elements of the arrays B, C, and D for its frequency values. Thus we can associate the three notes of a chord with a single key.

After the chord is played, the computer passes on to a new CALL KEY statement in Line 85Ø. Lines 86Ø-88Ø check the status variable (Y) for the new CALL KEY. IF no key is pressed, THEN control passes to Line 89Ø, which plays a millisecond of silence. This turns off the sound of the previous CALL SOUND with a negative duration value. Control then passes to Lines 9ØØ-95Ø, which change the color of the permanent graphics display and then returns to Line 76Ø for a new CALL KEY.

IF a new key is pressed, THEN Line 87Ø, which checks the status variable of the second CALL KEY, is true; control goes back to Line 79Ø, which starts the search loop to evaluate the return variable of the new note. If this return variable turns out to be one of the lucky 18, another chord will be played; if not, then silence. IF the key is held down, THEN Line 8ØØ is true and the CALL SOUND continues.

By using two CALL SOUND statements with negative durations, one playing chords and the other playing silence, we enable our organ to play the chord as long as the designated key is being pressed. When the key is released, the sound becomes silence and the color of the graphics display changes. The conditional statements make all this possible by constantly monitoring the keyboard.

Link 7

Link 7 does not exist as a program line, yet it is in every line of a program. In fact, it's the most important link of all. Without it, there would be no program. Link 7 is you, the programmer.

Computers exist for one reason, so people can use them. For all their machine-like logic, the programs that people create, are really extensions

of the human mind. If you've gotten this far, then congratulate yourself because you now have the understanding and the tools to make the computer work for you. Whatever that work may be, we wish you the best.

21 Your Home Computer As A Terminal

Although your TI-99/4A is a powerful tool by itself, a vast new world opens for you when you use it as a terminal to interact with larger computers or computer networks. Electronic information utilities such as The Source and CompuServe maintain huge data storehouses. Your humble computer can give you access to hundreds of specialized databases on topics from agriculture to zoology. Many computer networks offer services such as electronic mail, financial reporting, travel reservations and scheduling and bulletin boards. If you can get an account with a university computer, you'll have access to new programming languages, memory space, and even printing facilities. The possibilities for enhancing the way you learn and do business are limited virtually only by your imagination.

To use the 99/4A as a terminal you'll need several pieces of equipment. The first is the Terminal Emulator II, a plug-in software module sold by Texas Instruments, that bypasses the main microprocessor in your computer so it can act as a terminal. Next you will need a telephone **modem** for "talking" with other computers. Because the data used by computers are not compatible with signals normally sent over telephone lines, a modem *mod*ulates computer signals into a form compatible with telephone lines and *dem*odulates these signals back into a form the computer understands. Last, you'll need an **RS232 peripheral controller interface**. A modem is a computer peripheral and cannot be controlled by your 99/4A without this interface.

The speed of data transmission via a modem is measured in bits per second, or **baud**. Ordinary phone modems operate at either 300 or 1200

baud but the Terminal Emulator will allow you to use only a 300 baud modem. This translates roughly into a transmission rate of about 30 bytes, or keyboard characters, per second.

Phone modems can either be **acoustical** or **direct-connect**. Acoustical modems feature a cushioned receptacle for the handset of your telephone; so the signals passing in through the phone speakers can be "heard" by the modem. A direct-connect modem is directly wired into both computer and phone line. Direct-connect 300 baud modems can be purchased for as little as $100. The modem you buy should be compatible with your RS232 interface. The general name for the appropriate modem is the Bell 103 type. These devices are widely available, although not all come with cables specifically adapted to the 99/4A.

Currently you have several possibilities for getting an RS232 interface. You can buy the peripheral expansion box made by Texas Instruments for the 99/4A and get the RS232 circuit card that goes with it. You can also buy one of the RS232 interfaces built for the 99/4A by other companies; these dock directly into your computer without the peripheral expansion box. (Manufacturers of these products advertise in the *99'er Home Computer Magazine.*) You can also buy the TI Hex-Bus peripheral system, comprising a Hex-Bus adapter that docks into your main console and the HEX-Bus RS232 that connects to the adapter. Although the Hex-Bus peripheral system has been on TI's retail price list for some months, the units were not available at the time of this writing.

Your 99/4A can communicate with other computers because its keyboard is ASCII encoded. (ASCII is an acronymn for American Standard Code for Information Interchange.) When you press a key on your keyboard, the collection of bits produced and transmitted over the telephone line will be recognized as the same key by any other ASCII-encoded computer system. In addition to an ASCII keyboard, your computer also contains ASCII text formatting commands that allow you to format text on a computer system. A list of these commands can be found in the Appendix in Section Three of your *User's Reference Guide.*

When you are sending or receiving data over the telephone lines, you are using **serial** transmission: all the bits making up the data are transmitted in a line, one behind the other. Data can also be transmitted in **parallel**, but only where there's a group of parallel lines to carry all the bits in a particular byte simultaneously, as in a ribbon cable hooked up to a printer. Ordinary telephone lines cannot transmit in parallel.

Synchronous and asynchronous are the two major communication techniques. Asynchronous communication exists when each character is transmitted at random intervals of time such as those which exist when hitting the keys of a keyboard. Asynchronous transmission takes place more slowly than synchronous transmission which requires that data be sent in lock-step as regulated by a clock. Your computer will deal exclusively with asynchronous transmission.

Computer systems use an error-checking mode known as **parity.** When a byte composed of seven bits is sent out over the line, an extra bit is added to it to achieve this purpose called the **parity bit.** The parity bit will either be a "1" or a "0" depending on whether even parity or odd parity is being used. In even parity, the parity bit is changed to either a "1" or a "0" in order to create an even number of "1s" in that byte. Similarly, odd parity seeks to create an odd number of "1s" in a given byte by adding either a "1" or a "0." Your Terminal Emulator II will ask you to choose which parity to use. Familiarize yourself with the requirements of the system you are using.

The Terminal Emulator II will also ask you whether your host system uses simplex, half-duplex, or full-duplex transmission. Simplex transmission works in one direction only. Half-duplex transmission is two way, but only one way at a time, and full-duplex transmission is simultaneous two way transmission.

Most databases and information utilities charge you for the time you're "on line." Several factors—including the desirability of the database and the time of day—affect these charges. Like ordinary telephone service, database fees are often cheaper at night. Most databases and information utilities operate in conjunction with telecommunications networks that provide local phone numbers for users. Some information services include the charges for these communications networks in their fees; others break their billing into separate charges.

Two widely known communication networks are Telenet and Tymnet. Before you open an account with a database or information utility, find out if there's a local network number on their system which can help you save money on your phone bills.

If you're a smart shopper, you should be able to get the necessary components for using your 99/4A as a terminal for $300 or $400. If you also have a printer or a disk drive, the Terminal Emulator will enable you to copy the information you receive. Although advanced closed networks using synchronous transmission or a communication code other than ASCII won't be available to you, the huge and growing store of resources now ready for your 99/4A makes the price of admission well worth paying.

22 System Options

Selling presently for a mere $100 (after rebate), the TI-99/4A is indeed a remarkable bargain. But before you can use it in your business, hook it up as a terminal, or do word processing, you will have to add some equipment to the system. This chapter describes some of the hardware and software options you might try.

EXTENDED BASIC

Now that you're becoming comfortable with the TI BASIC that comes with your machine, you may wish to add the Extended BASIC Command Module. This plug-in module adds about 40 new commands, extending your computer's power and flexibility. Among the extra commands are those that control **sprites,** smoothing-moving graphics images.

Extended BASIC allows you to put more than one statement on a program line and is more memory-efficient. Programs in Extended BASIC run faster than those in "normal" TI BASIC, and many tasks that are either tedious or impossible in plain BASIC can be done with Extended BASIC. Moreover, most of the best third-party software is written in Extended BASIC, making the module well worth the price.

SPEECH SYNTHESIZER

The speech synthesizer is a small brushed aluminum unit that docks directly into the main console. It can synthesize speech from external software or from keyboard input. Speech synthesizers for microcomputers used to cost several hundred dollars, but during 1982 and 1983, Texas

Instruments is running a promotion that gives you a free synthesizer with purchase of six software modules.

PERIPHERAL EXPANSION BOX

For major expansion of your 99/4A computer system you need the Peripheral Expansion Box. This solidly built silver box is actually larger than the main console and contains a fan, power supply, a motherboard with slots for seven peripheral cards, and space for either one full-size or two half-size disk drives. (Disks, also called diskettes, are flat, circular pieces of plastic with magnetic surfaces that, like tapes, can store programs. Disk drives are analogous to the tape recorder, allowing the computer to transfer information to and from the disk.)

The box connects to the console via a cable that plugs into the connecting port on the right side.

RS232 Card

The RS232 card, a peripheral control interface that fits into the peripheral expansion box, controls devices like printers and telephone modems. It comes with one serial and one parallel port, although you can buy an inexpensive device to convert the one serial port into two. Thus outfitted, the RS232 card can control three separate devices.

Disk Drive Controller Card

You need a disk drive controller card to connect your computer to a disk drive that "reads" and "writes" information from or on a disk. Some disk drives have two read-write mechanisms, called heads; others have only one. The controller card can control up to four double-sided disk drives, each storing about 180K bytes of information. It takes up one slot in the peripheral expansion box.

32K Memory Expansion Card

The 32K memory expansion cards adds 32,768 bytes of memory to your computer. You'll need this extra memory if you wish to use sophisticated systems such as UCSD Pascal, LOGO or the Editor Assembler (described later). This card also takes up one slot in the peripheral expansion box.

P-Code Card

The P-code, or pseudo-code, card is the hardware component of the UCSD p-System™ (see the section on the UCSD p-System in this chapter).

The P-code card also takes one slot of the peripheral expansion box.

Disk Drives

If you plan to do serious work with large amounts of information, such as word processing or accounting, you really need disk drives—ideally, you should get two. Disks can store far more information than tapes, and disk drives transfer the information to and from the computer much faster than tape recorders. Furthermore, disk drive read/write heads can go directly to a piece of information anywhere on a disk; tapes, in contrast, must be read sequentially from the beginning.

The 99/4A uses 40-track ANSI (American National Standards Institute) standard 5¼ inch floppy disk drives manufactured by several companies. ("Floppy" refers to the disk, not the drive.) The drives may be either one-sided (reading only one side of a disk) or double-sided and comes in two sizes—full-size or the new space-saving half-size. One-sided disks store about 90,000 bytes of information; double-sided disks store twice that much.

Printers

Printers are perhaps the most popular and useful computer peripherals. Printers print by two methods: dot-matrix or daisy wheel. Dot-matrix characters are printed as individual dots grouped together to form a letter, number or symbol. Daisy wheel printers put characters on the page much as does a conventional typewriter, one fully formed character after another. The characters sit at the ends of little type bars arranged just like the petals of a daisy. Daisy wheels are interchangeable to give you different type styles.

A printer is controlled through either the serial or parallel port of the RS232 card depending on how it receives information. Parallel printers are usually faster than serial versions because all the bits in a particular byte arrive simultaneously at the printer through an interconnecting ribbon cable. You can connect a printer to your computer through either the serial or parallel port of the RS232 card.

Another option in obtaining a printer is to use an electronic typewriter which is capable of being interfaced with the RS232 card or carries its own RS232 interface. The problem most often encountered in connecting printers to the computer is in acquiring, or creating, the appropriate connecting cables. With such cables any printer compatible with the RS232C standard will work with the peripheral expansion box system.

Telephone Modems

The telephone modem is the other commonly purchased computer peripheral. It is used to allow your computer to communicate through the phone lines to other computers or computer networks. For a fuller explanation of this topic, see Chapter 21, "Using Your Home Computer As a Terminal."

The Fully Configured System

The fully configured TI-99/4A computer system contains most of the devices discussed to this point. You can always add new software, operating systems, or peripherals, but the fully configured system gives you a full-bore computer capable of professional work.

ALTERNATIVES TO THE PERIPHERAL EXPANSION BOX SYSTEM

You don't necessarily have to buy all your equipment from Texas Instruments for it to fit inside the peripheral expansion box. Several independent companies now manufacture and sell compatible RS232 and memory expansion cards. These companies also manufacture RS232 and 32K memory expansion peripherals that dock directly into the main console without the peripheral expansion box. Such products can help reduce system costs if you just wish to run a printer or modem or to use LOGO. If you want the full power of the system, however, you can't avoid the peripheral expansion box because TI currently makes the only disk drive controller card.

WORD PROCESSING

For efficient word processing work, you need a fully configured system, minus perhaps the modem and speech synthesizer. You could do without one disk drive, but a single drive can be rather inconvenient.

Several software companies currently put out word processing packages in Extended BASIC, and some packages in machine language are also beginning to appear. TI WRITER, a word processing module produced by Texas Instruments, is currently the most powerful and sophisticated word processing software for the 99/4A.

Microsoft Multiplan™

Microsoft Multiplan is a highly sophisticated second- generation elec-

tronic spreadsheet program. Marketed by Texas Instruments as a software module, it is widely regarded as more extensive and flexible than the popular VisiCalC™ spreadsheed program.

UCSD p-System™

The UCSD p-System™ is a powerful operating system for your 99/4A that provides an alternative to the original 99/4A operating system. You need the p-System to run the UCSD version of the computer language Pascal—more powerful and much faster than TI BASIC. The UCSD p-System™ consists of a P-code card that fits inside the peripheral expansion box and several software diskettes. You may purchase these components separately or as a package.

An important advantage of the UCSD system is that it has been cross-compiled for different microprocessors. In other words, software written in the UCSD system for one computer can be run on a different computer with little or no modification. (Normally, programs written for different microprocessors in different machines are incompatible.) Thus software written in the UCSD p-System™ for an Apple II could run on your 99/4A while it is using the UCSD system with little modification.

This kind of compatibility gives you access to a wide variety of software, so we would expect use of the p-System on the 99/4A to grow. But as of summer, 1983, using the p-System has not been common, and Texas Instruments has not supplied much software. If you'd like to use the UCSD p-System™, check the "Resource List" in Chapter 23 for a group to contact.

LOGO

LOGO is a computer language designed primarily for those who have little or no computer experience and has become popular in elementary and secondary school education. Texas Instruments has been involved in LOGO's development since the beginning and markets a version for the 99/4A as a software module. Running LOGO requires 32K memory expansion.

Machine Language

This is the native language of the 9900 microprocessor and as such is the most powerful language the computer can use. Machine language is the language used in the solid state software modules for your 99/4A.

Texas Instruments makes several products that let you work with machine language by writing in assembly language; the main one, the

Editor/Assembler, is available as a software module. The mini-memory module also lets you to work with assembly language plus giving you an extra 4,000 bytes of mini-memory. Contained within the module and supported by battery power, the mini-memory retains its contents even when detached from the main console. Many people are becoming interested in assembly language for the 9900, and more third- party software written in this language is appearing.

FORTH

FORTH is another fast and powerful language available for the 99/4A on diskette from Texas Instruments. FORTH requires the Editor/Assembler and 32K expansion memory. A third party version of FORTH on disk is now also available.

Voice Recognition

Milton Bradley will produce a peripheral for the 99/4A called the Expander, which can recognizes voices and enables you to speak to your computer. The device comes with a set of headphones, a microphone, and its own multi-position keypad; software is being developed. The Expander plugs into the joystick port.

Winchester Hard Disk

For those not satisfied with floppy disk drives, a Winchester hard disk information storage peripheral is being sold by Myarc, Inc. (P.O. Box 140, Basking Ridge, NJ 07920). This device comes in two models that can store either five million or ten million bytes and is faster than floppy disk systems. Myarc's hard disk system comes complete with built-in software.

23 Resource List

We've compiled in this chapter some of the major contacts and organizations which can help you make the most of your home computer by providing you with advice, information, and services.

Texas Instruments Customer Relations Hotline (800-858-4565)

This toll-free hotline provides answers to home computer rebate inquiries, gives out exchange center locations for repairs, and answers minor technical questions. In addition, marketing literature and referrals to other TI company contacts are mailed from this location. Hours are 8:00 a.m. to 4:15 p.m. Monday through Thursday and until 3:15 p.m. on Friday, Central Standard Time.

Texas Instruments Software and Peripheral Hotline (800-858-4075)

You can place retail orders for hard-to-find software and peripherals at this number. The staff will answer basic questions, mail marketing literature, and make referrals to local retail outlets. Hours are 8:00 a.m. to 4:15 p.m. Monday through Thursday and until 3:15 p.m. on Friday, Central Standard Time.

User's Group Coordinator

Updated lists of TI-99/4A User's groups are periodically published in the Texas Instruments *Home Computer Newsletter.* If you can't locate the group in your area, or are thinking of starting one of your own, contact: Texas Instruments, User's Group Coordinator, Mr. Ed Wiest, P.O. Box 10508, Mail Station 5890, Lubbock, TX 79408.

Technical Assistance Line (806-741-2663)

These people can help with technical problems, especially those about interfacing your computer with peripherals.

Repair and Service Contacts

You can find the location of your local repair and service center by calling the Customer Relations Hotline. These centers require you to appear in person. Mail-in service is done only at the following address: Texas Instruments Repair Service, 2303 N. University Dr., Lubbock, TX 79415

Computer Advantage Clubs (800-858-4069)
In Texas (800-692-1318)

Sponsored by Texas Instruments in a number of major metropolitan areas around the country, the Computer Advantage Clubs offer introductory computer courses for children and adults, including programming in BASIC and LOGO, word processing, and Microsoft Multiplan. Call the numbers above to get registration information and find out about the courses being held in your area.

Texnet℠ (800-336-3330)

Texnet is a unique service specifically designed for 99/4A owners which is part of The Source℠ electronic information utility. Many services are available to 99/4A owners under Texnet, including the ability to "borrow" 99/4A software over the wire. Call the toll-free number above for information.

International 99/4 Users-Group, Inc. (405-787-8521)

This is by far the oldest and most sophisticated independent 99/4A support organization. The group has around 70,000 members and a software library exceeding 1,500 programs. The group publishes the *Enthusiast 99* magazine and acts as a mail-order retailer of 99/4A software and peripherals. Membership costs $12/year; a presidential membership, which includes special privileges, is $50/year. Contact: International 99/4 Users-Group, P.O. Box 67, Bethany, OK 73008.

99/4 Users of America (313-736-3774)

This organization puts out a newsletter and sells TI software to its

members at reduced rates. They maintain a programming assistance line (at the number above) between 2:00 p.m. and 4:00 p.m., Eastern Standard Time. The group sells software from a number of third party sources and maintains a directory of programs. Dues are $20/year. Contact: 99/4 Users of America, 5028 Merit Dr., Flint, MI 48506.

99'er Home Computer Magazine

This is the largest and most highly developed periodical available to the 99/4A owner. It contains many articles on educational applications, entertainment, operating systems, software, peripherals, interfacing, programming and much more. The magazine also carries the largest cross section of advertisers for TI home computer hardware and software found in one place. Subscriptions in the U.S.A. are $25, $45, and $63 for one, two, and three years respectively. Contact: *99'er Home Computer Magazine,* P.O. Box 5537, Eugene, OR 97405.

Young People's LOGO Association

Those interested in using TI LOGO should consider contacting this group. YPLA produces two publications; *Turtle News* and *LOGO Newsletter* and is actively involved in creating a network of local organizations active in using LOGO. Membership is free to those under 18 years of age. Contact: Young People's LOGO Association, 1208 Hillsdale Dr., Richardson, TX 75081.

UCSD Pascal Systems User's Society

This organization sponsors a special interest group for Texas Instruments computer users operating the UCSD Pascal System. The group has a large volume of software available to members. Membership is $20/year. Contact: UCSD Pascal System User's Society Secretary, P.O. Box 1148, La Jolla, CA 92038.

Computer and Software News

Although not focused on specific computer system, this publication is an excellent source of microcomputer market news and often carries developments in the 99/4A market before they are noted elsewhere. Subscription is $18/year. Contact: *Computer & Software News,* A Division of Lebhar-Friedman, Inc., 99 Park Ave., New York, NY 10157.

EASY PROGRAMMING WITH THE TI-99/4A

TI 99/4A Users: Learn to *create your own programs*!

If you want to go beyond "canned" software . . . If you are frustrated with unexplained exercises . . . If you are ready to explore what you can *really* do with your computer . . . Then, this book is for you!

You don't have to be an experienced programmer to use EASY PROGRAMMING FOR THE TI-99/4A. Now that you're familiar with the fundamental techniques of BASIC programming, this book will lead you into advanced professional techniques. It gives practical applications, plus such resource listings as: *international user groups, toll-free computer service telephone numbers, and magazines and other resources for the TI user.*

"From space shuttles and satellites to junk mail lists and bank statements, computers are all around us. Learning state-of-the art programming techniques for your own home computer provides you with an added understanding of this important tool. Computers are an extension of the people who use them, just as the programs you write are extensions of your own imagination."

So don't be afraid—learn to create!

$10.95

ISBN 0-8176-3166-6