

By the Best Selling Computer Author of *Timelost*
Joseph C. Giarratano, Ph.D.



Texas Instruments
99/4A

BASIC GUIDE



Computext

TEXAS INSTRUMENTS
99/4A
BASIC Guide

by
Joseph C. Giarratano, Ph.D.
Computext, Inc.
Indianapolis

Copyright © 1983 by Computext, Inc.
Library of Congress Catalog Card Number: 83-72569
ISBN 0-913847-00-3

Cover Art: Carpenter Graphics
Word Processing: Sue Lee, Inc.
Typeset by Alexander Typesetting, Inc.
Printed by: Fotolab, Inc.
Manufactured in the United States of America

Published by Computext, Inc.
P. O. Box 50942
Indianapolis, IN 46250

DEDICATION

This book is dedicated to my family: Jane, Jenna, Melissa, and Anthony.

ACKNOWLEDGEMENTS

I greatly appreciate the assistance of Al Lovati, Gene Harvey, Sally Catena, Fred Kitson, Bill Stacy, Sue Lee, Roger B. Crampton, Gary Tinker, Karen Brantingham, and Sue Dismore for their reviews of the book. Thanks to Ed Drake, Bill Barniea, John Yantis, Percy Clark, and Don McHolland for their support and encouragement. A special note of thanks is due Beverley Hasenbalg for her excellent editing and comments.

I also appreciate the loan of a SelectaVision Video Monitor from RCA.

OTHER BOOKS BY JOSEPH C. GIARRATANO INCLUDE:

Foundations of Computer Technology

Computer history, software, printers, terminals, and integrated circuits

Modern Computer Concepts

Memory devices, disks, tapes, data communications, teletext, videotex, and a guide to purchasing a computer system. The above two-book set makes a thorough introduction to computer technology for the beginner and forms the basis of computer literacy.

BASIC: Fundamental Concepts

BASIC: Advanced Concepts

The above two books on BASIC provide a complete introduction to BASIC for Microsoft BASIC, and Digital Equipment Corp. BASIC. You will learn the pitfalls in converting programs written in one dialect of BASIC to another and gain a much better understanding of BASIC since the book's programs are written for both Microsoft and Digital Equipment Corp. BASIC. Application programs include exact precision arithmetic of 200 digits or more, internal storage in BASIC; PEEK and POKE, with an application to a self-modifying program for graphing functions; roundoff-error in business calculations.

Timex/Sinclair 1000 User's Guide, Vol. I

Timex/Sinclair 1000 User's Guide, Vol. II

Timex/Sinclair 1000 Dictionary and Reference Guide

The above three books provide complete coverage of Timex/Sinclair BASIC with many examples to business, education, games, and personal use.

Timelost for the Timex/Sinclair 1000

Timelost for the TI-99/4A

Timelost for the Atari

Timelost for the VIC-20

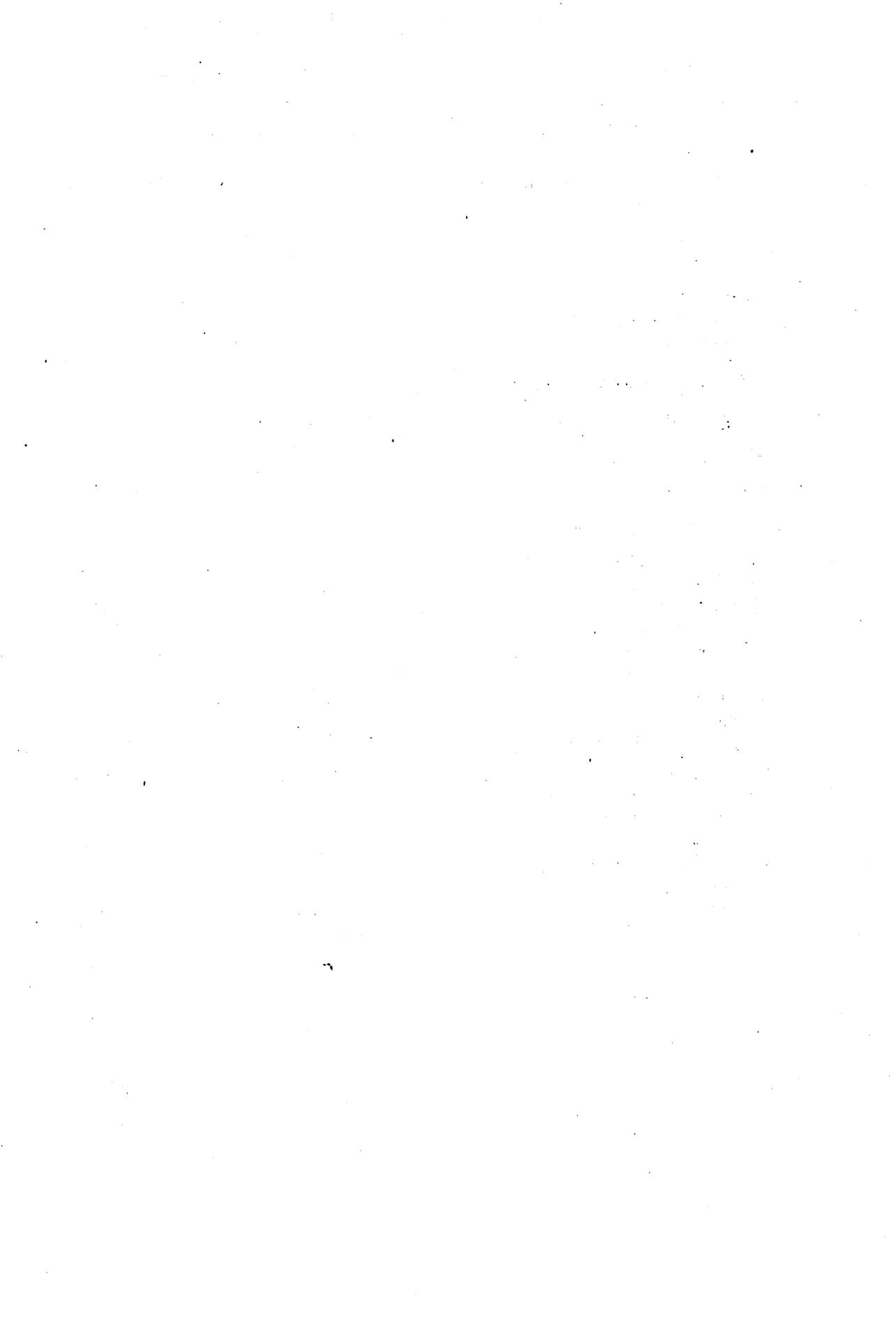
The above four books have a computer adventure story set in comic book form. Follow the adventures of a teenage computer whiz, his kid sister, and their mysterious friend from the future as they battle the evil Warlord of the Universe. Follow the story and then enter the programs listed in the book which match the action in different scenes. All programs are clearly documented and meant to explain to you how each program was written. You are encouraged to enhance the programs and to come up with your own versions of the games.

Contents

CHAPTER 1—Getting Started	1
Playing Around	1
Don't Call for Repair Yet	2
What's My Function?	3
An Exception to the Rule	3
Adding Things Up	4
Multiplication and Division	6
Very, Very, Big and Teeny, Tiny Numbers	8
Powerful Powers	10
Mixing Things Up	11
A Matter of Priority	12
Getting Spaced Out	14
Stringing Along	16
CHAPTER 2—Giving You the Power	19
Tabbing—TAB	19
Square Roots—SQR	21
Random Numbers—RND	24
RANDOMIZE	25
Integer—INT	27
Absolute Value—ABS	29
Sign—SGN	30
Exponential—EXP	30
Natural Logarithm—LOG	31
Sine—SIN	32
Cosine—COS	33
Tangent—TAN	34
Arctangent—ATN	34
CHAPTER 3—Beginning Programming	35
Give Me A Line	35
Running With It	35
Listing	36
Making Life Easy on Yourself	37
How to Change Your Number	39
A Little More LIST	41
What's NEW?	42
Easy Number	42
Ups and Downs	45

CHAPTER 4—Let’s Get Variable	47
A Shoebox Called MONEY	47
Bad MONEY	48
Stringing Along	50
Give Me Cents	52
How Much Can You Store in a String?	53
CHAPTER 5—Giving Input	55
Easy Input	55
Let’s Make Money	60
How to Make More Money	62
How’s Your IRA?	63
Can You Afford It?	64
CHAPTER 6—Going Automatic	67
Go To It	67
Let’s Continue	68
Caught in a Loop	69
Getting Control	69
Let’s Be Logical	71
If This Is True	74
Guess My Number	77
The Average Way	79
Control Your Loop!	80
How to Make a Nest Without Birds	82
Jumping In and Jumping Out	83
Building Tables	86
CHAPTER 7—Getting to Know Your Dimensions	89
Names, Names, Names	89
It’s All in the Loop	90
Let’s Get Dimensioned	91
Give Me a Name and I’ll Tell You the Number	92
Add Some Strings	94
Read My Data	96
Well, Tape My Data	99
Many Dimensions	104
Changing Your Option	107
CHAPTER 8—Debugging and Documentation	109
Ignore This Sign	109
A Good Design	111
Follow That Line!	113
Stop That Line	114
O.K., Break It Up	114

CHAPTER 9—Stringing Along	117
What A Character	117
What's The Opposite of a Character?	118
How's My Length?	118
Give Me Your Position	119
Strings And Numbers	120
The Value of a String	121
Cutting Up a String	121
How To Get Organized	122
Scrambled Animals	125
CHAPTER 10—Economize, Economize	129
Going Down Under	129
Let's Get Drilled	131
How's Your Memory	133
Get Defined	134
CHAPTER 11—Give Me A Call	137
Clearing Things Up	137
Getting Keyed Up	137
Stop That Data!	140
Put It Where You Want	141
Draw Me a Picture	144
Play Me a Tone	148
Update Your Display	150
CHAPTER 12—Let's Get Graphic	155
What a Character	155
R Walk	158
Change Your Screen	162
Add Some More Color	163
What's That Character?	166
Alpha Pilot	166
Illusions	171
The Great Adventure	177
APPENDICES	181
GLOSSARY	197
INDEX	213



Chapter 1

Getting Started

In this chapter you'll learn how to start using your computer. You'll see how to enter information into the computer and how the computer gives you back the answers. Also, you'll learn what to do if you make a mistake. You'll see how to use your computer as a calculator using BASIC (Beginners All-purpose Symbolic Instruction Code). BASIC was first developed in 1964 at Dartmouth College as a way of helping students easily learn to program. It has proven to be a popular and easily-learned computer language. Millions of people ranging from children to senior citizens have learned to use BASIC and versions of BASIC are available on every home computer today. Using BASIC, you'll find out how to use your computer to easily arrive at the answer to complicated problems. In fact, this ease of use is what makes computers so popular. You just tell the computer what to do, and it does all the hard work of the calculations. Of course, computers have many more applications than just doing math. In later chapters, you'll learn how to write programs in BASIC and see applications to business, education, and games.

Playing Around

Assuming you've hooked up the computer according to the Texas Instruments (TI) instruction manual, then the first thing you should do with your new computer is—play around with it. Get the feel of the keys and how they operate.

Let's get started by turning on the computer and pressing any key. Also, be sure the key labeled "Alpha Lock" is pressed down. This will give you all capital letters on the screen and so make it easier for you to read the screen. Now press the "1" key for TI BASIC. You should see the message

TI BASIC READY

at the lower left corner of your TV screen. There should also be a blank line under the message and below it a greater than sign ">" followed by a blinking black square. The ">" is called a **prompt** because the computer expects you to type something in, such as a command or other information. Note that words in **boldface** are in the Glossary. This blinking square is called a **cursor** and indicates where the next letter you type will appear on the screen.

Let's give the computer some **input**. The term input means the data or other information that goes into the computer. Tap the "1" key and you'll see a "1" appear at the bottom left of your screen. Notice that the cursor has now moved over one space to the right. Now hold down the "1" key for a couple of seconds and you'll see a whole group of "1"'s appear on the screen. If you momentarily press and release a key, you'll get a single key printed on the screen. But holding down a key gives many repetitions of the same character. This is called an automatic repeat or **auto repeat** feature of your computer and saves you effort in typing.

What happens if you keep pressing down the key? Let's try it and see. Hold down the "1" key and you'll see four rows of "1" 's appear. Then the cursor stops at the far right and you'll hear a tone from the computer. This tone indicates that the BASIC is designed to accept only four rows of "1" 's. How many "1" 's is that? Well, if you count all the one's in a row, you'll see there are twenty-eight "1" 's in a row. So you can input a maximum of $4 \times 28 = 112$ characters.

Now try pressing any other letter or number key on the keyboard. These are called the alphanumeric keys. The term **alphanumeric** is a contraction of alphabetic and numeric. The numeric keys are all on the top row of the keyboard. From left to right, they are 1 through 9 and then a \emptyset . Notice that the \emptyset is printed on the keyboard with a slash through it. This is a common convention so that you don't confuse the letter O with the number zero. If you press the letter O key, you'll see it printed on the screen as a rectangle with straight edges while the number zero is printed with rounded edges. You'll see that the last "1" on the bottom row changes to whatever key you press. Also, press the spacebar and the last character on the screen will become a blank. This shows that the spacebar gives the blank character. Other keys such as the "/" (division sign), "=" (equal sign), and ";" will give those characters when you press them.

The alphanumeric and punctuation keys are called **printable** keys because their images are printed on the screen. Other keys, such as ENTER, SHIFT, FCTN, CTRL, and ALPHA LOCK, do not produce a visible character on the screen. However, the computer does know that the key has been pressed.

Also, just like a typewriter, you can access the top symbol on a key by first holding down the SHIFT key and then the key you want. For example, hold down the SHIFT and then press the key with the "+" on it to get the "+" printed.

Don't Call for Repair Yet

If the computer is displaying something on your TV screen and you don't press a printable key for about 9 minutes, something strange happens—the TV screen goes blank. Don't call for repair yet. Chances are that the computer has used one of its built-in features called the screen-saver feature. If you haven't used any key for awhile, the computer blanks out the screen image. This prevents any possible damage to the screen. If the

same image is displayed for a long time, it's possible that the image may get burned into the screen. Although it's very unlikely, your TI computer is designed to blank out the image so that it can't happen.

To get back your original image, just press one of the printable keys. If the image still doesn't come back, then you may want to have your computer checked.

What's My Function?

Suppose that you don't like all those "1"'s on your screen. How can you change them? Your computer has several features which allow you to easily change your input. This is called **editing**.

If you look down at the bottom right of the keyboard, you'll see a key labeled **FCTN**. This is an abbreviation for the word "function". The function key is very useful when used with certain other keys. Just as a shift key on a typewriter expands the functions a key can perform, so too does the FCTN key. In fact, you can think of FCTN as an additional shift key.

Notice the left pointing arrow on the front side of the "S" key. The FCTN key activates this arrow to move the cursor left. In fact, you'll notice that many keys have symbols on their front side. All these symbols are accessed with the FCTN key. So the FCTN key allows the same key to have more than one function.

For example, hold down the FCTN key and then press the "S" key. You'll see the cursor move back. Keep holding down these keys and you'll see the cursor move all the way to the left and then jump up to the line above. If you keep holding down these keys then eventually the cursor will stop at the first number "1" that you input. By just holding down FCTN and tapping the "S" key, you can make the cursor move one character at a time.

Now that you've seen what the left arrow on the "S" key does, you can easily guess that the right arrow on the "D" key does. To see if your guess is right, hold down the FCTN key and press the "D". You'll see the cursor move toward the right. However, the up arrow on the "E" key and the down arrow on the "X" key will not make the cursor jump up or down a line. These keys have a different purpose which we'll discuss in a later chapter.

Now try using the FCTN key to print the characters on the front side of keys. Notice that the "U" key gives an underscore, `_`, when pressed. To get the minus sign, `-`, you need to press SHIFT and the key with the `"-"` and `"/` on it, to the right of the "P" key. If a key doesn't have a symbol on its front side, the FCTN key gives only a blank character when you press FCTN and the key.

An Exception to the Rule

However, there's an exception to this rule. The top row of keys have special functions that are not printed on the keys. To see what their func-

tions are, just insert the strip that came with your computer into the track above the number keys. You should have

DEL	INS	ERASE	CLEAR	BEGIN	PROC'D	AID	REDO	BACK	QUIT
!	@	#	\$	%	^	&	*	()	+
1	2	3	4	5	6	7	8	9	0 =

Let's find out what these keys do. First type in 12345. If you still have all the "1"'s on your screen from the first example, press the ENTER key. The computer will print the message

* BAD LINE NUMBER

which you can just ignore. We'll discuss line numbers in a later chapter on programming. Right now, the main thing you should see is that pressing the ENTER key moves the cursor back to the left of the screen and starts you off fresh. Now type in the numbers 12345. Then hold down the FCTN key and press the "S" key to move the cursor back to where it's positioned over the 3 on the screen. Then press the FCTN and "1" keys momentarily. You'll see the "3" disappear and all you'll have left is "1245" on the screen. The cursor is now positioned over the "4". What's happened is that the FCTN and "1" key perform a deletion of the character that the cursor is positioned over. That's why the strip that you inserted has a "DEL" for deletion. The character is deleted and those characters to the right move left to fill up the deleted character's position.

You may have guessed that INS stands for insertion. To see how insert works, hold down the FCTN key and press the "2" key. Nothing visible happens on the screen because you haven't yet told your computer what to insert. But the computer is now ready to insert characters starting at the cursor. To see this, release the FCTN key and press the "3" key. You'll see the original "12345" displayed again. Press the "3" key again and you'll see another "3" appear as "123345". In fact, you'll keep on inserting characters at the cursor position as long as the insertion feature is activated.

You can cancel the insertion feature by simply moving the cursor using the FCTN key and right or left arrow keys, or by pressing the ENTER key. Just move the cursor one or more positions and insertion is cancelled. For example, move the cursor one position to the right and press the "6" key. Notice that the "6" replaces the number at the cursor position. The "6" was not inserted because the insertion feature had been cancelled when you moved the cursor.

Now let's try the ERASE key. Use the FCTN and "3" key to activate this feature. You'll see that the whole line is now erased.

The CLEAR feature of the "4" key is used in programming and we'll discuss it in more detail in a later chapter. If you just press the FCTN and "4" key, any information on the screen just moves up one line. To see this, type in XXXXX and then press FCTN and the CLEAR keys. Hold down these keys and you'll see the "XXXXX"'s move up the screen. We describe this as saying the screen is **scrolling** up. Notice that the information on the top line moves off the screen when scrolling occurs.

The BEGIN, PROC'D (proceed), AID, REDO, and BACK features are used by certain games and programs. The meaning of those keys depends on how they are interpreted by those programs. However, the PROC'D feature has one interesting effect. When the screen saver feature is activated and the image disappears from your set, you have to press a key to bring back that image. The key you press would appear on the screen. You may not want to have that key image appear on the screen, but only to regain the original picture. It turns out that you can press the FCTN and "6" keys to activate the PROC'D feature to bring back the screen image. The nice thing about this is that no key image appears on the screen and no scrolling takes place. The PROC'D key just allows you to proceed without changing the screen image.

The final special function key we'll discuss is QUIT. Activate this and you'll see the original Texas Instruments message appear on the screen. The QUIT key is similar to turning off the power in your computer and then turning it on again. You start off fresh and everything you've done is erased.

Adding Things Up

As a first step, let's see how to add $2 + 2$. Enter the following command to your computer. This tells the computer to add $2 + 2$ and print the result on your TV screen.

```
PRINT 2+2
```

To enter the **PRINT**, you must spell out the letters of PRINT using the keyboard keys. Also, to get the "+" sign, hold down the SHIFT key and then press the key in the upper right hand corner labeled with a "+". If you make a mistake entering this, just use the function keys to delete or to insert the right characters.

Notice that the computer doesn't do anything after you entered the PRINT 2+2. The reason it's not doing anything is because the computer doesn't know you're finished giving input. Perhaps you wanted

```
PRINT 2+2+2
```

or

```
PRINT 2+2-2+2
```

There is a very important key that tells the computer you're finished with input. This is the **ENTER** key located on the right end of the middle row of keys. Press this key and you'll see the screen scroll up. The answer of 4 will appear below the "R" of "PRINT". The result from the computer is call **output**. From now on, you'll have to remember to press the ENTER key when you're done with input.

Now try

```
PRINT 2-2
```

Notice that the minus, “-”, is a shifted key on the right end of the second row of keys from the top. After you press ENTER, you’ll see a zero printed under the R. Let’s try a few more examples as shown below. Also shown below each PRINT are the answers as you would see them on the screen.

```
PRINT 2+2+10
```

```
14
```

```
PRINT 100+100
```

```
200
```

```
PRINT 7879-13
```

```
7866
```

```
PRINT 81+92-19-34+17
```

```
137
```

```
PRINT -99+3-64-12
```

```
-172
```

Notice that in the last example, the minus sign is printed under the “P” of print. The computer always leaves room for a minus sign even if one is not needed. That’s why the answers are printed starting in the second column from the left. The leftmost column is reserved for the sign of the answer. A blank is printed for positive numbers and a minus sign is printed for negative numbers.

Of course, your computer is not limited to calculations with whole numbers or integers. It can also deal with decimal numbers. For example, try

```
PRINT 8.5+2
```

```
10.5
```

where the decimal point is on the key next to the right SHIFT key. Also try

```
PRINT 1.5+1.5-30.3
```

```
-27.3
```

Multiplication and Division

As you’d expect, your computer also knows how to multiply and to divide. However, the computer uses the asterisk, “*”, as the symbol for multiplication, rather than the common “X”. Try the following

```
PRINT 2*2
```

where the “*” is a shifted key over the “8”. You’ll see a 4 appear under the “R” of PRINT. Try some more examples, as follows:

```
PRINT 10*10
```

```
100
```

```
PRINT 123*650
```

```
79950
```

```
PRINT 7891.35*871.2
6874944.12
```

```
PRINT -900.6*868.111111
-781820.8666
```

As you can see from these examples of multiplication, the computer does not print commas to separate the output. Likewise, the computer won't accept commas separating numbers either.

The last example above also points out the limitation of the computer. The exact answer is actually

```
781820.8665666
```

However, the computer is designed to calculate and to print numbers only to a certain precision. Internally the computer calculates numbers to 13 or 14 digits. This internal result is rounded off to the ten *most significant digits* and printed. The most significant digits of a number are the highest digits. For example, the three most significant digits of 8653.781 are 865. Different computers are designed for different degrees of precision. For example, in many other versions of BASIC, six or nine significant digits are common. If you want more than the built in precision, you'll need a program. As an example, see the book *BASIC: Advanced Concepts*. This includes programs to add, subtract, and multiply numbers with 200 or more digits of precision. The programs are written in Microsoft BASIC, and Digital Equipment Corp. BASIC. These versions can be easily converted to run on your computer.

Now let's try some division. Try

```
PRINT 4/2
2
```

Where the division sign, "/" is the key on the right end of the second row from the top. Now try

```
PRINT 100/10
10
```

```
PRINT 999/3
333
```

```
PRINT 8621.356/.2
43106.78
```

```
PRINT -2603.7/.001
-2603700
```

```
PRINT 1/3
.3333333333
```

Note that the last example shows again that the computer shows only the ten most significant digits of the result. The fraction 1/3 is actually the

never-ending decimal number .333333 . . . where the dots at the end mean the 3's go on forever.

Now try

```
PRINT 1/3*3
```

and you'll see the result of 1. printed on your screen. Does this mean the result is exactly 1? Let's find out by having the computer subtract $1/3*3$ from 1. The answer should be 0.

```
PRINT 1-1/3*3
1.E-14
```

Very, Very, Big and Teeny, Tiny Numbers

The answer of 1.E-14 is not zero. It is a very small number of

```
.0000000000000001
```

Your computer expresses very large and very small numbers in a notation called **scientific, power-of-ten, or exponential notation**. Notice all the zeroes in front of the 1 above. Rather than having you write all these zeroes, just imagine that the decimal point moves 14 places to the right and then write this as

```
1E-14
```

Here are some other examples of this notation.

Number	Power-of-ten	Exponential Notation
1	1×10^0	1E0
.1	1×10^{-1}	1E-1
.01	1×10^{-2}	1E-2
.001	1×10^{-3}	1E-3
.0001	1×10^{-4}	1E-4
.00001	1×10^{-5}	1E-5
.015	1.5×10^{-2}	1.5E-2
.000396	3.96×10^{-4}	3.96E-4

Each number is shown expressed as a power-of-ten. The meaning of the minus sign of the exponent is also shown below

$$10^{-1} = 1/10$$

$$10^{-2} = 1/10^2 = 1/(10*10) = 1/100$$

$$10^{-3} = 1/10^3 = 1/(10*10*10) = 1/1000$$

The parentheses around the $10*10$ means that this calculation is done before the division into 1.

For example, try these on your computer. Use the "E" key for E.

```
PRINT 1E0
1
```

```
PRINT 1E-1
.1
PRINT 1E-2
.01
PRINT 1E-3
.001
PRINT 1.5E-2
.015
PRINT -8.62E-6
-.00000862
PRINT 1E-10
.00000000001
PRINT 1E-11
1.E-11
```

Notice that numbers less than $1E-10$ are automatically expressed in exponential notation by the computer.

Just as minus or negative exponents are used to express small numbers, positive exponents are used to express big numbers. However, you don't have to write the "+" before the exponent. When you leave out the sign, the computer assumes you mean "+".

Try the following

```
PRINT 1E1
10
PRINT 1E2
100
PRINT 1E3
1000
PRINT 1E4
10000
PRINT 3.986E6
3986000
PRINT 9.999999999E9
9999999999
PRINT 1E10
1.E+10
```

Notice in the last two examples that numbers up to and including 9999999999 can be printed without exponential notation. but numbers greater than this, such as $1E10$, are automatically printed in exponential notation.

Now try

```
PRINT -2.3E4
-23000
```

```
PRINT -2.3E-4
-.00023
```

Notice what a big difference a negative power makes in the magnitude or size of a number.

Using exponential notation, your computer can work with numbers as large as 9.999999999999999E127 and as small as 1E-128

Powerful Powers

Besides the standard arithmetic operators of multiplication, division, addition, and subtraction, your computer can also perform **exponentiation** or raising a number to a power. The customary way of writing a number raised to a power is with a superscript. For example

$$1 = 10^0$$

$$10 = 10^1$$

$$100 = 10^2$$

Here the number 10 is called the **base** and the power it is raised to is called the **exponent**. This is the same as the exponential or power of ten notation we've discussed. On the computer the caret symbol " \wedge " is used for exponentiation. Try

```
PRINT 10^0
1
```

```
PRINT 10^1
10
```

```
PRINT 10^2
100
```

```
PRINT 10^9
1000000000
```

```
PRINT 10^10
1.E+10
```

In the last two examples, notice that 10 raised to the 9th power can be printed as 1 followed by 9 zeroes. But 10 raised to the tenth power is automatically expressed in exponential notation because it's too big.

Small numbers can also be calculated. Try

```
PRINT 10^-1
.1
```

```
PRINT 10^-2
.01
```

```
PRINT 10^-10
.0000000001
```

```
PRINT 10^-11
1.E-11
```

You can also raise numbers to a base other than 10, where the base is the number which is raised to the power. For example, try

```
PRINT 2^2
4
```

```
PRINT 3^3
27
```

```
PRINT 51.3^-2.1
.0002563029
```

```
PRINT -2^-2
-.25
```

In the last example, 2 is first raised to the minus second power. This is $2^{-2} = 1/2^2 = 1/4$

and then negated

$$-1/4 = -.25$$

Mixing Things Up

You can easily compute the answer to combinations of these arithmetic operators. The term **arithmetic operator** means a symbol for a

	Example
exponentiation, ^	2^3
negation, -	-2
addition, +	$2+2$
subtraction, -	$2-2$
multiplication, *	$2*2$
division, /	$2/2$

Notice that the symbol for negation, i.e. making negative, is the same as the symbol for subtraction. The minus sign is used for both negation and subtraction. The negation sign is called a **unary operator** because it operates on the one term which follows it. The prefix “un” comes from the Latin word “unus” meaning one. For example, try

```
PRINT -2
```

and you’ll see a -2 printed. Now try

```
PRINT --2
```

and you'll see a 2 printed. The negation sign has made a negative 2 into a positive 2. However, the minus sign used in subtraction is called a binary operator because it requires two operators. The prefix "bi" means two while the term operand means the term that an operator acts on. The prefix "bi" comes from the Latin word "bis" meaning two. For example, try

```
PRINT 3.1^2+10/2.1-2*37.0004
-59.62889524
```

```
PRINT 3*4+18/2
21
```

```
PRINT 3*12*2/4+6.1-9.02
15.08
```

```
PRINT -1.5*-2
3
```

In the last example, note that the computer is multiplying a negative 1.5 times a negative 2. The minus signs cancel and so the answer is a plus 3.

Now try

```
PRINT -1.5*2
-3
```

```
PRINT 1.5*-2
-3
```

In this case, when a positive number is multiplied by a negative number, the result stays negative. An analogy is that if you owe \$1.50 and your debt is doubled, then you owe twice as much.

A Matter of Priority

A group of arithmetic operators and operands is called an **arithmetic expression**. For example, $2 + 2$ is an arithmetic expression. When the computer calculates the number equivalent to this expression, the computer is evaluating the arithmetic expression. In evaluating an arithmetic expression, the computer follows certain rules. For example, in a previous example

```
PRINT 3*4+18/2
```

the computer first multiplied 3 by 4 to get 12. It then divided 18 by 2 to get 9. Then it added the 12 and 9 to get the final answer of 21. This is the natural way that people do arithmetic, so BASIC is designed to evaluate arithmetic expressions in this way. The computer does not first add 4 to 18 to get 22, divide this by 2 to get 11 and then multiply 11 by 3 to get 33.

The computer does arithmetic according to the following order of priority for the arithmetic operators

exponentiation
negation, also called unary minus
multiplication and division
addition and subtraction

So when the computer looks at an arithmetic expression, it first performs exponentiation, then negation, then multiplication and division, and finally addition and subtraction. If operators have equal priority, the computer evaluates them on a left to right basis. For example

```
PRINT 2*2/4+3  
4
```

The computer multiplies $2*2$ to get 4, then divides $4/4$ to get 1, and adds $1+3$ to get the final answer of 4.

To show that negation comes below exponentiation, try

```
PRINT -2^-2  
-.25
```

The computer first raises 2 to the minus 2 power, giving .25. Then it performs the negation of .25 to yield the answer of -.25. If negation had a higher priority than exponentiation, the computer would have raised minus 2 to the minus 2 power and the result would have been .25 instead of the -.25 you see. Actually, your computer also accepts the positive equivalent of negation. For example, try

```
PRINT -+2  
-2  
PRINT +-2  
-2
```

The + sign is a unary operator. However, it doesn't do anything and so is not of much use in calculations.

You can use parentheses to force the computer to evaluate an expression any way you want. Try

```
PRINT (-2)^-2  
.25
```

Notice the difference in the answer. In this case, we are raising minus 2 to the minus 2 power to yield a plus .25.

As another example, without parentheses

```
PRINT 2*3-4  
2
```

but you could use parentheses to change this to

```
PRINT 2*(3-4)  
-2
```

In evaluating this, the computer first finds the number 2 and then the * indicating multiplication. Then it finds the left parenthesis, and then 3-4. It then finds the right parenthesis and evaluates 3-4 as -1. The computer then multiplies the first 2 by -1 to give the answer of -2.

Parentheses have the effect of increasing the priority of the operations inside of them. The innermost parentheses are performed first. For example

```
PRINT 1+(2*(3+4/(8-4)))
9
```

In evaluating this expression, the computer evaluates the innermost expression in parentheses first, and so calculates 8-4 as 4. It then divides 4 by 4 to get 1. Then it adds 3 + 1 to get 4. Next, it multiplies 2 by 4 to get 8. Finally, it adds 1 to 8 to yield the final answer of 9.

If you're in doubt as to how the natural priorities will cause an expression to be evaluated, you can do one of two things. First, you can try a sample calculation to see what the result is. You can then use parentheses if necessary to change the order. Second, you can always use parentheses and rely little on the natural order.

The problem with using parentheses is that the computer does take longer to evaluate expressions with parentheses. The computer must store the intermediate results of parentheses and then retrieve those results later. This delay will not be noticeable if you're just doing a single calculation. But in a program where the same calculation may be done thousands of times, the delay can become noticeable.

Getting Spaced Out

Suppose you want to do two calculations, such as

```
PRINT 3*4+82
```

and

```
PRINT 98.3*206.05
```

Rather than your having to type out PRINT each time, you can space out your calculations using commas. Try

```
PRINT 3*4+82,98.3*206.05
94                    20254.715
```

If you count over the columns from the left, you'll see that the leading 2 of 20254.715 is in column 16 where the leftmost column you can type in is column 1. However, printing of a number always starts with its sign. So the second result was actually printed starting in column 15 since that's where the sign would be printed.

For example, try

```
PRINT 1,-1
```

and then enter the digits 1234567890123456 (but don't press the ENTER key). The reason for entering all these numbers is so that you can easily tell what columns the results are printed in. As you can see, the minus sign of the second 1 is indeed printed in column 15. A group of 14 columns is called a **field**. Now erase this long number. Then try

```
PRINT 1,1,1
```

and you'll see the third "1" printed in column 1 of the next line. Now try

```
PRINT 1,1,,1
```

The third 1 is now printed starting in the second field of the second row. As you can see, since there was nothing in between the commas, then nothing was printed in the first field of the second row. However, the computer did space over according to the commas.

Now try

```
PRINT 1;1
1 1
```

As you can see, the second "1" is started in column 4 for its sign. To show this more clearly, try

```
PRINT -1,-1
-1 -1
```

While you know that space is reserved for the sign, why is there another space after the first "1"?

The reason is that BASIC also reserves space for a decimal point after the number. Even if no decimal point is needed, such as with integers, a space is still allocated.

As you can see, the effect of the semicolon in separating numbers is to force their printing as close as possible. This is convenient when you have a lot of numbers to print, since commas allow only two results per line while semicolons allow much more.

You can also print on different lines by using the colon to tell the computer to start printing on the next line. For example, try

```
PRINT 1:1:1:1
1
1
1
1
```

You can use multiple colons to skip lines. Try

```

PRINT 1:1::1:::1::::1
1
1
1
1
1
1

```

Notice that if a colon follows a colon, a line is skipped.

Stringing Along

Your computer can print letters, punctuation marks and other printable symbols on its keyboard. Just put them between quotes so that the computer knows they are symbols. Try the following, where you access the quote by using the FCTN and "P" key. Remember to press the ENTER key after the last quote.

```

PRINT:"HELLO": "12345"
HELLO
12345

```

In the second example above, notice how putting a colon before the "HELLO" makes the computer skip a line before starting to print output.

All the characters, punctuation symbols, numerals and other printable symbols between quotes are called a string of characters. A **string** is just a group of symbols arranged in a pattern, like "HELLO" or "1000 MAIN STREET" or "+-*;,@!". A string like "HELLO" may have meaning to a human reader, but does not have any special meaning to the computer. Strings are also called literals. A **literal** is just a symbol with no special meaning to the computer. The computer does attach a special meaning to strings like PRINT and other words of BASIC.

It's important to realize that there are numerals before "MAIN STREET" and not a number. That is,

```
"1000"
```

is a string of numerals or symbols. The computer prints them exactly as shown. For example, try

```

PRINT "1000"
1000

```

but

```

PRINT 1000
1000

```

Notice that in the second example the computer thinks you want to print the number 1000, so it starts printing with a leading blank because 1000

is a positive number. However, when you put quotes around 1000, the computer thinks you want to print the string of numerals and does not put a leading blank. A string is printed exactly as you give it to the computer, while the computer reserves a leading blank or minus sign for a string of numbers, also called a **numeric string**.

Just as you can combine numbers by addition with the “+” sign, there is an operator that allows you to combine strings. Try

```
PRINT "12"&"34"  
1234
```

The ampersand, “&” is a shifted “7” key, and is used to combine strings end to end. This combination of strings end to end is called **concatenation**. Also try

```
PRINT "SMITH,"&"JOHN"  
SMITH,JOHN
```



“That's the last time you ever say my budget isn't balanced.”

Chapter 2

Giving You the Power

Now that you've learned how to use your computer as a simple calculator, you're ready to get more power. Your computer has many built-in functions that will make life easy for you in the most demanding applications. You may not need all of this power, but we'll cover it in enough detail so that you'll become familiar with their use. Most of all, you'll see the limitations as well as the power of your computer. The most useful functions for the example programs in this book are, TAB, SQR, RND, RANDOMIZE, ABS, and INT. You may want to read about these functions carefully and just skim through the discussion of the other functions.

Tabbing—TAB

The **TAB** acts like the **tabbing** of an ordinary typewriter. TAB allows you to start printing anywhere on a row. Try these

```
PRINT TAB(1);1
```

```
1
```

```
PRINT TAB(5);1
```

```
1
```

```
PRINT TAB(10);1
```

```
1
```

```
PRINT TAB(27);1
```

```
1
```

```
PRINT TAB(28);1
```

```
1
```

In each of these examples, you supply the name of the function, TAB, and then its argument. The **argument** of a function is the value you supply to the function. The computer then prints the output using the argument as input to the function. The functions of the TI-99/4A consist of the name of the function followed by the argument within parentheses. Some functions which we'll see later can have more than one argument in parentheses.

The tabbing is done starting from the leftmost column, which is column 1. In the first example, the sign of the number would be printed in

column 1 under the “P” of PRINT. Since a positive number is printed with a leading blank for a plus sign, a blank is printed in column 1 and the magnitude of the number in column 2. The **magnitude** of a number is its value without respect to its sign. For example, the magnitude of 2 is 2, and also the magnitude of -2 is 2. The magnitude is also called the absolute value of a number.

In the fourth example of TAB(27), the magnitude of the number is printed in the 28th column of the screen since its sign is printed in the 27th column. In the fifth example, you’ll notice that the computer has started over by printing the number’s sign in column 1 and the magnitude in column 2. If you look closely, you’ll see there is another difference between a TAB(1) and a TAB(28). The computer has skipped a row in printing the 1 for the TAB(28) example.

Now try

```
PRINT TAB(27);10
```

```
10
```

and you’ll see the 10 printed starting under the “R” after a row has been skipped. Why does PRINT TAB(27);1 do a print on the row underneath, but PRINT TAB(27);10 skip a row? The answer is that the computer checks ahead to see how many characters are to be printed on the line and then decides if there is enough room to print them all on the line. If there is not enough room to print all the characters, the computer skips a line and starts the first column of the next line. The idea behind this is to avoid breaking up a number so part of it will be on one line and the rest of it on the next line. For example, if you wanted to print the number 1010, you wouldn’t want it to be printed as

```
10
```

```
10
```

which might be confusing to you.

However, as with everything else in life, there are exceptions. Try

```
PRINT TAB(29);10
```

```
10
```

In this case, you see that the computer did not skip a line. The reason this occurs is that if the argument of the function is greater than 28, the computer divides the argument by 28 and then tabs over the remainder. Dividing 29 by 28 gives 1 so the computer tabs by 1. The remainder is accepted by the computer if it is a number between 1 and 28. For example, try

```
PRINT TAB(57);1
```

```
1
```

You’ll see the computer start printing in column 1 because the remainder of 57/28 is 1.

But try

```
PRINT TAB(56);1
```

```
1
```

and you'll see the 1 printed after the computer skips a line. The reason this occurs is that $56/28$ equals 1 plus a remainder of 28. This remainder of 28 is acceptable to the computer and so it does not divide the 28 by 28.

As you saw in a previous example,

```
PRINT TAB(28);1
```

```
1
```

does skip a line because the computer does not want to break up the digits of a number by printing them on separate lines.

You can use multiple TAB's in one PRINT command. For best results, use semicolons to separate the TAB's and times to be printed. For example

```
PRINT TAB(5);1;TAB(10);2
      1      2
```

Now let's put together everything we've learned and print a graphics design. Note that the following PRINT command shows what you will see on the screen as you type in the statement. Just keep typing in the symbols. The computer will automatically move the cursor down a line as you keep typing. After you type the last quote of "HAPPY HOLIDAYS", press the ENTER key.

```
PRINT TAB(14);"*":TAB(13);"*
      **":TAB(12);"*****":TAB(14);
      "":TAB(7);"HAPPY HOLIDAYS"
              *
              ***
              *****
              :
      HAPPY HOLIDAYS
```

As you can see, the PRINT, TAB and colon allow you to create all kinds of designs.

Square Roots—SQR

You can easily calculate the square root of a number using the **square root function, SQR**. Try the following examples

```
PRINT SQR(36)
```

```
6
```

```
PRINT SQR(50)
```

7.071067812

From the first example above, you can immediately see that 6 is the square root of 36, since $6 * 6 = 36$. But, what about the second example? Let's check it out. Enter

```
PRINT 7.071067812*7.07106781
2
50.
```

Also try

```
PRINT SQR(50)*SQR(50)
50.
```

Note that the presence of a decimal point after the 50 indicates the answer is very close, but not exactly 50. That is, the computer calculation does not give an integer result for this calculation. However, if you try

```
PRINT 25*.4
10
```

the result is an integer, 10, because there is no decimal part of 10 within the 13 significant digits calculated by your computer. That is, the computer calculates $25 * .4$ as

10.0000000000000

and so prints the integer 10. Can we get a more precise square root of 50? Remember that your computer calculates to 13 or 14 significant digits, but only prints the most significant 10. Let's see one of those hidden digits by subtracting the integer part, 7, from the square root of 50. Enter

```
PRINT SQR(50)-7
.0710678119
```

From this you can see that a more accurate value of the square root of 50 is 7.0710678119 rather than the 7.071067812 we got by taking the square root directly.

Can we get an even more precise answer, say 13 significant digits? Sure. Just subtract the most precise value we have for the square root from the square root of 50.

```
PRINT SQR(50)-7.0710678119
-3.5E-11
```

The answer of -3.5E-11 shows the final digits of precision calculated by your computer. So the most precise square root of 50 is

```
7.0710678119
-0.000000000035
7.071067811865
```

So 7.071067811865 is the square root of 50 calculated to 13 significant digits. As another way of showing this, try

```
PRINT SQR(50)-7.071067811865
```

```
0
```

If you try

```
PRINT 50-7.071067811865*7.07
1067811865
7.E-12
```

you'll see that the square root we printed is still not exactly the square root of 50.

However, the difference of 7.E-12 is smaller than the magnitude of $-3.5E-11$ we got with 7.0710678119 as the square root of 50. This indicates that 7.071067811865 is closer to the square root of 50 than 7.0710678119. The computer can't calculate the square root of 50 exactly because it is a multiple of the square root of 2 and can't be expressed in a finite number of digits. That's why you still can't get an exact integer when you multiply the roots. For example

```
PRINT 7.071067811865*7.07106
7811865
50.
```

The computer does not consider the integer "50" and the decimal number "50." as the same number. They are close, but not exact.

Of course, you can also use powers to calculate the square root, or any root of a number. For example, try

```
PRINT 50^.5
7.071067812
```

or

```
PRINT 50^(1/2)
7.071067812
```

Both give the square root of 50. However, note that

```
PRINT 50-(50^.5)*(50^.5)
-1.21E-10
```

has a bigger magnitude than the

```
PRINT 50-SQR(50)*SQR(50)
7.E-12
```

calculated by the square root function. For the most accurate result, use the square root function. Likewise,

```
PRINT 50^(1/3)
3.684031499
```

gives the cube root of 50. Because the TI-99/4A calculates to 13 or 14 significant figures, the results of SQR(50) and $50^.5$ are very close.

Random Numbers—RND

One of the most popular and fun applications of computers is playing games. In many types of games, it's necessary that the computer produce random numbers. A **random number** is one whose value is not known in advance by you. For example, you might want to make a game like dice. The computer must generate random numbers to simulate the rolling of dice. The word simulate means that the computer acts like something else. For example, the computer could **simulate** alien spaceships attacking your base, a card game, dice and many other things.

It's very easy to generate a random number using the random number function, **RND**. Enter

```
PRINT RND
.5291877823
```

Notice that the RND function does not have any argument. The RND returns a random number greater than or equal to 0 and less than 1. This can be written in the following way

$$0 \leq \text{RND} < 1$$

For example,

$$0 \leq .5291877823 < 1$$

The symbol, “<” means that the number to the left of “<” is less than the number to the right. The symbol “ \leq ” means less than or equal to. Another useful symbol is “>”. For example,

$$2 > 1$$

means that 2 is greater than 1. The “<” symbol is a shifted key over the comma, and the “>” is a shifted key over the period to the right of the comma.

Let's try some more random numbers.

```
PRINT RND
.3913360723
```

```
PRINT RND
.5343438556
```

```
PRINT RND
.3894551053
```

As you can see, these numbers are all different. Let's print out a few more, but save ourselves the effort of typing PRINT all the time. Enter

```
PRINT RND;RND;RND;RND
.2555008073 .5621974824
.2553391677 .5882911741
```

Notice how using a semicolon to separate items in a printlist really cuts down on the typing you must do.

RANDOMIZE

Now let's try an interesting experiment. Turn the power off your computer and then turn it on again. Get back into BASIC and type in the following. As before, the PRINT command is shown as it appears on your screen. Keep typing in the symbols and press the ENTER key after the last RND.

```
PRINT RND;RND;RND;RND;RND;RND;RND;RND;RND
D;RND;RND
.5291877823 .3913360723
.5343438556 .3894551053
.2555008073 .5621974824
.2553391677 .5882911741
```

If you compare these random numbers to the ones from the previous section on RND, you'll see they are the same.

What's going on? Well, it turns out that the random numbers generated by your computer aren't truly random. In fact, the proper name for these computer random numbers is **pseudorandom**. The prefix **pseudo** means false, so pseudorandom numbers are literally false random numbers. The pseudorandom numbers from your computer are calculated by a formula. It's really quite hard to come up with truly random numbers. In the interest of saving time for calculations, your computer uses a formula to generate these numbers.

Every time you power up your computer, the computer starts calculating these pseudorandom numbers from the beginning. So you always get the same sequence of pseudorandom numbers. Later on when we get into programming, you'll also see that the computer starts calculating the pseudorandom numbers from the beginning before every program is run. There are many thousands of different pseudorandom numbers that your computer can generate before it starts over again. For many games and applications, this is plenty.

However, your computer provides a function which will start off the random numbers at a different point in the sequence each time. This is the **RANDOMIZE** function, and like RND, it too has no argument. To see how RANDOMIZE works, turn your computer off and then back on. Now type

RANDOMIZE

and press the ENTER key. Now type in the following PRINT command and press ENTER after the last RND.

```
PRINT RND;RND;RND;RND;RND;RND;RND;RND;RND
D;RND;RND
.861119366 .849727399
.4897973528 .1733429849
.029304338 .1852545258
.5020166158 .2159153281
```

Chances are you will not get the random numbers shown above because the RANDOMIZE is starting the random numbers off at a different place in the pseudorandom number sequence. Try turning your computer off and on again, or just press FCTN and the “+” key (QUIT) to start over, and repeat this RANDOMIZE and PRINT commands. Each time, you’ll get different random numbers.

However, you can specify a certain starting place in the pseudorandom sequence by giving a **seed** or argument to RANDOMIZE. The argument of RANDOMIZE is called the seed because its value determines the following pseudorandom numbers. Unlike other functions, there is no need to put the seed in parentheses. Try

```
RANDOMIZE 1000
PRINT RND;RND
.3066028468 .262016067
```

Now enter the same commands over again.

```
RANDOMIZE 1000
PRINT RND;RND
.3066028468 .262016067
```

As you can see, giving the same seed to RANDOMIZE always starts off the same random numbers sequence.

You can even use arithmetic expressions for the seed. For example

```
RANDOMIZE 1+2
PRINT RND;RND
.3433917535 .6613386522
RANDOMIZE 3
PRINT RND;RND
.3433917535 .6613386522
```

Both RANDOMIZE 1 + 2 and RANDOMIZE 3 have the same seed.

You can use positive or negative numbers or decimal numbers such as .5 as the seed. However, there is one thing to watch out for. The computer does not use the entire value of the seed you give it.

Your computer actually converts the seed number to another form called a **binary number**. Most computers use binary numbers for calculations because of the components and circuits used in computers. Binary calculations are faster and binary computers use fewer components than computers designed to manipulate decimal numbers.

Binary numbers are composed of only the 0 and 1. The following table shows some ordinary binary numbers and their decimal representation.

Decimal Number	Binary Number
0	0
1	1
2	10
3	11
4	100

5	101
6	110
7	111
8	1000
9	1001
10	1010

Since the prefix “bi” means two, binary numbers use only the two numerals, 0 and 1. A binary digit of 0 or 1 is called a **bit**, which is a contraction of binary digit. A group of 8 bits is called a **byte**. Another common term is the **kilobyte** (K byte) which is 1,024 bytes. In fact, the memory of a computer is usually expressed in K bytes. For example, the standard TI-99/4A has 16 K bytes of memory.

Any decimal number that you supply the computer is internally represented by 8 bytes by a technique called the **normalized radix representation**. For more details on the internal storage of numbers, see your TI Users Reference Guide on accuracy, and *BASIC: Advanced Concepts*, p. 126. However, only the two most significant bytes are used for the seed value. Since only the two most significant bytes are used, not all the seeds you supply will be unique. Thus you might get the same random number sequence even though you specify a different value for the seed. For example, seeds from 0 to 100 are all unique. To show this, try a RANDOMIZE and PRINT RND with some seeds from 0 to 100. However, seeds from 100-199 are all the same. Likewise seeds of 200-299, 300-399 etc. are all the same up to a seed of 10000. Seeds from 10000-19999, 20000-29999 etc. are then the same.

If you're not sure that two seeds are the same, it's best to test it with RANDOMIZE seed and PRINT RND commands.

Integer-INT

The **integer function, INT**, is very useful in calculations. Try the following example

```
PRINT INT(1);INT(1.9);INT(2)
1 1 2
```

In this example, the integer function always returns the integer part of its argument. That is, the decimal or fractional part of the argument is thrown away.

Now enter the following example for negative arguments

```
PRINT INT(-.5);INT(-1);INT(-
2.5)
-1 -1 -3
```

As you can see, the integer function for negative arguments returns the integer that is smaller than the argument. For example, -1 is smaller than -.5 and is the next integer. A 0 would be greater than .5. Likewise, -3 is the next integer smaller than -2.5.

In general, the integer function returns the next smaller integer of its argument.

In most cases, the integer function works fine. However, you can run into trouble with numbers that are very close to an integer. For example, try

```
PRINT INT(1.999999999999)
1
```

This works all right and does return a 1. But including another 9 as in the following example

```
PRINT INT(1.9999999999999)
2
```

returns a 2 instead of the 1.

What's happened is that the computer converts 1.9999999999999 to a binary form before applying the integer function. The number we've supplied exceeds the computer's precision of about 13 digits and the computer interprets it as 2 instead of 1.9999999999999. Of course, you can write a computer program to deal with numbers having more than 13 digits of precision. For example, in *BASIC: Advanced Concepts*, programs are shown for multiplication, addition and subtraction of numbers with 200 digits of precision. These programs are shown for two other versions of BASIC, which are very similar to TI BASIC.

As an example of how useful the integer function is, let's use it to generate random numbers between 1 and 6. This could represent the numbers produced by a die as it's rolled in a dice game. Enter

```
PRINT INT(6*RND+1)
4
```

Of course, you may get a number other than 4 depending on what point you're at in the pseudorandom sequence.

If you keep printing out numbers as above, you'll see they all have values of 1, 2, 3, 4, 5, or 6.

There is a handy formula you can use to generate numbers between A and B where B is the smaller of the two.

```
INT((A-B+1)*RND+B)
```

For the die example above, $B = 1$ and $A = 6$ because we want to generate numbers between 1 and 6. Substituting in the above formula gives

```
INT((6-1+1)*RND+1)
= INT(6*RND+1)
```

which is what we used.

As another example of how useful the integer function is, let's use it to round off numbers. For example, suppose you want to round off a number to the nearest integer. Just add .5 to the number and apply the integer function. For example, to round off 1.4, enter

```
PRINT INT(1.4+.5)
1
```

To round off 1.9, enter

```
PRINT INT(1.9+.5)
2
```

The above works because the general rule for rounding is that if the decimal part of a number exceeds .5, return the next highest integer. So adding a .5 to a number and applying the integer function will return the next highest integer.

You can also use the integer function to round off numbers to a certain number of decimal places. For example, suppose you want to round off a number to one decimal place. Try the following example to round off 8.6345 to one decimal place

```
PRINT INT(10*8.6345+.5)/10
8.6
```

If you want to round off 8.634 to two decimal places, try

```
PRINT INT(100*8.6345+.5)/100
8.63
```

For rounding off to three decimal places, enter

```
PRINT INT(1000*8.6345+.5)/1000
8.635
```

The general formula for rounding off a number X to N decimal places is

$$\text{INT}(10^N * X + .5) / 10^N$$

For the example above where we rounded off 8.6345 to 2 decimal places, we have

```
PRINT INT(10^2*8.6345+.5)/10^2
8.63
```

This is the same as above since $10^2 = 100$. Likewise, for three decimal places, $10^3 = 1000$ and we would also get the same answer as the first time.

Absolute Value—ABS

The **absolute value function**, **ABS**, always returns a positive number. The absolute value is the magnitude of a number. Basically, the absolute value throws away any negative sign of its argument and returns

a positive number. If the argument is positive the absolute value just returns the number. Try

```
PRINT ABS(-2);ABS(2)
2 2
```

Sign—SGN

The **sign function, SGN**, returns the sign of its argument in the following way. If the number is positive, the function returns a value of 1. If the argument is 0, then a 0 is returned. If the argument is negative, then a -1 is returned. Try the following. As usual, the PRINT command is shown as it appears on your screen. Press the ENTER key after you type in the last right parenthesis “)”.

```
PRINT SGN(5);SGN(0);SGN(-2.5)
1 0 -1
```

Exponential—EXP

The **exponential function, EXP**, returns the base number of natural logarithms raised to the power of its argument. Just like pi or 1/3, the base of natural logs cannot be expressed exactly by a finite number of decimal digits. The base is commonly written as the letter “e” and is about 2.718281828. Like pi, the number symbolized by the letter “e” cannot be exactly represented by a finite series of digits.

As an example

```
PRINT EXP(1)
2.718281828
```

shows the ten most significant digits of “e”. For some more examples, try

```
PRINT EXP(0)
1
```

```
PRINT EXP(-.5)
.6065306597
```

```
PRINT EXP(10)
22026.46579
```

You can use an arithmetic expression as the argument. For example,

```
PRINT EXP(.5+.5)
2.718281828
```

```
PRINT EXP(2*4-8.5)
.6065306597
```

```
PRINT EXP(20/2)
22026.46579
```

Notice that you get the same results for the three examples above as for the first three examples because their arguments are the same. However, some arithmetic expressions may not be calculated exactly because of the finite precision of the computer and so there may be slight differences. For example

```
PRINT EXP(1E14*(2/3-1/3-1/3)
)
2.718281828
```

gives “e” instead of the correct answer of 1. Since $2/3 - 1/3 - 1/3 = 0$, then $1E14 * 0$ should equal 0 and $EXPO(0) = 1$. Instead, we get 2.718281828.

The problem we’ve run into arises because the computer only calculates to 13 or 14 decimal digits. For most calculations, this is fine. But some numbers like $1/3$ or $2/3$ can’t be exactly expressed as a finite number of decimals

```
1/3 = .3333333333333...
2/3 = .6666666666666...
```

where the three dots at the right mean the digits keep on repeating forever. Enter

```
PRINT 2/3-1/3-1/3
1.E-14
```

and you’ll see the result is not 0 but 1.E-14. So when this number is multiplied by 1E14, the exponents cancel and the argument of EXP is 1. Since $EXP(1) = 2.718281828$, you now see why we got the wrong answer.

The important thing to learn from this is that just like any other machine the computer has limitations. Just because it can do millions of calculations a second doesn’t mean they’re correct. You supply the intelligence—the computer does the work.

Exponentials and natural logs are inverse functions of one another. For example

$$1 = EXP(0) = EXP(LOG(1))$$

where LOG is the natural logarithm function. Likewise, for any number, N

$$N = EXP(LOG(N)).$$

Natural Logarithm—LOG

The **natural logarithm function, LOG**, returns the natural log of its argument. The log of a number is the exponent to which the base must be raised to give the number. For example, 2 is the log of 100 in base 10 because $100 = 10^2$. The number 10 is the base. It is raised to the power. Try the following examples

```
PRINT LOG(1)
0
```

```

PRINT LOG(2)
.6931471806

PRINT LOG(10)
2.302585093

PRINT LOG(2*2.5+5)
2.302585093

PRINT LOG(628.5)
6.443336028

```

Notice that you can use an arithmetic expression as the argument. Both $2*2.5+5$ and the 10 arguments return the same log.

The natural log returns the log to base “e” = 2.718281828 (see the discussion of the exponential function, EXP). To find the log of a number N in another base, B , use the formula

$$\text{LOG}(N)/\text{LOG}(B).$$

For example, the log of 20 in base 10 (**common logs**) is

```

PRINT LOG(20)/LOG(10)
1.301029996

```

where we’ve substituted $N=20$ and $B=10$ in our formula.

If you try

```

PRINT LOG(0)

```

you’ll get a

```

* BAD ARGUMENT

```

error message because the log of 0 is negative infinity, and the computer can’t compute it.

Sine—SIN

The **sine function, SIN**, returns the sine of an angle. For a right triangle, the sine is the ratio of the side opposite an angle to the hypotenuse. The angle must be in radians. There are $2*\pi$ radians in 360° . So 1 radian = $360/2*\pi$. To convert from degrees to radians, just multiply the degrees by $\pi/180$. Use 3.141592654 for π , or .01745329251944 for $\pi/180$. For example, to convert 30° to radians, use either

```

PRINT 30*3.141592654/180
.5235987757

```

or

```

PRINT 30*.01745329251944
.5235987756

```

The second example using .01745329251944 is more accurate than the first since more significant digits are used. A more precise value for pi to 14 significant figures is

$$\pi = 3.1415926535898.$$

Using this for pi will give the same answer as the second example.

Rather than your having to remember all these numbers or to look them up, there's an easier way to get pi. Enter

```
PRINT 4*ATN(1)
3.141592654
```

The ATN(1) is the arctangent function (see the section on the arctangent function). It turns out that 4 times the arctangent of 1 radian is pi. So to convert 30° to radians, use

```
PRINT 30*4*ATN(1)/180
```

or just

```
PRINT 30*ATN(1)/45
```

since $4*ATN(1)/180 = ATN(1)/45$. The answer in both cases above is .5235987756.

To get the sine of 30° , enter

```
PRINT SIN(30*ATN(1)/45)
.5
```

As another example, the sine of 90° is

```
PRINT SIN(90*ATN(1)/45)
1
```

Cosine—COS

The **cosine function, COS**, returns the cosine of an angle. For a right triangle, the cosine is the ratio of the side adjacent to the angle to the hypotenuse. Theoretically, the sine of 90° is exactly 1. However, the value returned by the sine function is very close. If you try

```
PRINT SIN(90*ATN(1)/45)-1
```

you'll get

```
-1.E-14
```

Just as for the sine function, the argument of the function is in radians. Try these examples

```
PRINT COS(30*ATN(1)/45)
.8660254038
```

```
PRINT COS(90*ATN(1)/45)
0
```

where the first example is for 30° and the second is for an angle of 90° .

Tangent—TAN

The **tangent function, TAN**, returns the tangent of an angle. For a right triangle, the tangent is the ratio of the side opposite the angle to the side adjacent the angle. Like the sine and cosine, the argument of the tangent function is in radians. Try these examples

```
PRINT TAN(30*ATN(1)/45)
.5773502692
```

```
PRINT TAN(90*ATN(1)/45)
```

```
* WARNING:
NUMBER TOO BIG
9.99999E***
```

The first example for 30° works all right. However, the tangent of 90° is infinity, which causes the error message of a number too big. In this case, the number exceeds

```
9.999999999999999E127
```

and the computer displays this number in the form

```
9.99999E+**
```

Since the computer can only show two digits for the exponent, it indicates an exponent greater than 99 by the two asterisks.

Arctangent—ATN

The **arctangent function, ATN**, is the inverse of the tangent function. The arctangent returns the angle in radians when it is given the tangent. Try these examples

```
PRINT ATN(.5773502692)
.5235987756
```

You'll see this is the angle in radians for 30° since

```
PRINT 30*ATN(1)/45
.5235987756
```

Chapter 3

Beginning Programming

While your computer makes a great calculator, that's just a small fraction of what it's capable of doing. The thing that really distinguishes a computer from a calculator is that the computer can carry out its instruction automatically. The computer is said to be following a program of instructions. A **program** is a group of instructions that the computer executes in a certain order. The program is stored in the **random-access memory (RAM)** of your computer. The RAM is memory that you can put programs and data in. The computer has another type of memory called **read-only memory (ROM)** whose contents can only be read. For example, the BASIC language of your computer is in ROM. The advantage of ROM over the RAM is that information in ROM is not lost when power is removed from the computer. In this chapter you'll learn to start programming.

Give Me A Line

When you wanted to use your computer as a calculator, you just typed in the command and pressed the ENTER key. The computer immediately did the calculation and output the result. This is called a direct or immediate command.

However, when you enter a program you don't want the computer to execute each command as you type it in. Instead the command is stored in the computers RAM memory. Commands are stored in a certain order in the memory. Each command is prefixed by a line number. The combination of a line number and command is called a **statement**. Also, press the ENTER key after each statement. As an example of a simple program, type in the following two statements.

```
1 PRINT 2+2
2 PRINT "HELLO"
```

Running With It

Now that this program has been entered into the RAM of your computer, we want the computer to **execute** the program. The term execute means the computer performs the instructions in the program. When the computer executes a program, it starts by doing the command with the lowest line number, then the next highest line number and so forth.

To execute a program, type in the command **RUN** and then press the ENTER key. The RUN command makes the computer begin executing the program starting from the lowest line number. You'll see the screen change color to green and then the output appear.

```
RUN
4
HELLO

** DONE **
```

Also notice that the symbol ">" does not appear when the computer is outputting. The ">" only appears when the computer is expecting you to input something. In the output above, the computer has printed the sum of 2+2 and then printed "HELLO". Notice that the computer has executed each statement in turn starting from lowest to highest line number. When no more statements are present in RAM, the computer stops execution after line 2 and prints the message "DONE".

Now enter the command
RUN 2

and this time only the word "HELLO" is output. When you follow the command RUN with a number, the computer starts execution with that line number. So in this case, the computer starts execution with line 2.

If you type RUN again, you'll get the same output as the first time—a "4" and then "HELLO". In fact, you'll get the same output every time this program is run.

For this particular program, our input data is fixed. The computer always adds 2+2 and prints "HELLO". In a later chapter, you'll learn how to change the input data while a program is running. This makes programs much easier to use.

Listing

Your computer has a very useful command which lets you see the program in memory. Type in the **LIST** command

```
LIST
```

and press the ENTER key. The program stored in memory will be listed on the screen.

Now try

```
LIST 1
```

and you'll see only line number 1 listed. Also, try LIST 2 and only line 2 will be listed.

Suppose you want to print 10*10 right after the PRINT 2+2 of line 1. Just enter

```
2 PRINT 10*10
```

When you list the program, you'll see

```
LIST
1 PRINT 2+2
2 PRINT 10*10
```

Notice that by entering in a new line with the same line numbers as a line stored in memory, the new line has erased the old one. RUN this program and you'll see

```
RUN
4
100
** DONE **
```

The original line 2 that printed "HELLO" is indeed gone from memory and no "HELLO" is printed. Let's restore the original line by entering it as

```
3 PRINT "HELLO"
```

When you LIST this program you'll see

```
LIST
1 PRINT 2+2
2 PRINT 10*10
3 PRINT "HELLO"
```

and when you RUN, you'll get

```
RUN
4
100
HELLO
** DONE **
```

The above technique is one way of editing a program. Type in the new line and then retype in the lines you had erased. Of course you must change their line numbers if you've inserted a line in between.

Making Life Easy on Yourself

One simple way of making life easy on yourself is to allow room to change your mind. Instead of entering lines with numbers 1, 2, 3, . . . , use line numbers that are multiples of 10. For example, if the original lines had been

```
10 PRINT 2+2
20 PRINT "HELLO"
```

then you could have used any line number from 11 to 19 and that line would be inserted between 10 and 20 automatically.

Let's try this and see how it works. First, turn off your computer's power and then turn it back on again or press the FCTN and QUIT keys. Get into BASIC and do a LIST. Notice that no program is listed. Instead you just get the error message

```
* CAN'T DO THAT
```

because there is no program in memory. The RAM memory in your computer is said to be a **volatile** memory type. The term volatile means that the contents disappear when power is removed. So one way of erasing the program in memory is to turn power off. A second way is to use the FCTN and QUIT keys, which also erase the contents of memory.

Now that the lines are gone enter

```
10 PRINT 2+2
20 PRINT "HELLO"
```

and RUN it. You'll see the same output as when the line numbers were 1 and 2. When the computer executes a program, it goes from smallest to highest numbers and doesn't care what values those numbers are. Now type in

```
10
```

and press the ENTER key, and then type in

```
20
```

and press the ENTER key. Now type in the LIST command and press the ENTER key. You'll see the message

```
* CAN'T DO THAT
```

because you've deleted all the program lines in memory and so the computer can't list them. You can use any line numbers for the program. For example, type in

```
525 PRINT 2+2
7968 PRINT "HELLO"
```

and the output would be the same. Now delete lines 525 and 7968 and type in the original lines

```
10 PRINT 2+2
20 PRINT "HELLO"
```

Let's insert a line now. Enter

```
15 PRINT 10*10
```

While it may appear to you that the line number 15 was not inserted because it stays after 20, let's see what the computer thinks. Enter the RUN command and you'll see

```

RUN
 4
 100
HELLO

```

```
** DONE **
```

Notice that the result of $10*10$ was indeed printed between the "4" and "HELLO". Do a LIST and you'll see

```

LIST
10 PRINT 2+2
15 PRINT 10*10
20 PRINT "HELLO"

```

When you add lines by typing them in, the output on the screen is not automatically updated. The line 15 you entered was changed in the computer's memory but the TV display of old output was not updated. After you make changes in a program, it's a good idea to do a LIST and confirm them. For example, enter this

```
5 PRINT "TI-99/4A"
```

and do a LIST. You'll see the updated listing come scrolling up the screen with line 5 as the first line of the program. When you run the program, you'll also see the output of line 5 first, as it should.

Now suppose you want to delete certain lines. Just enter in the line number of the line you want to delete and press the ENTER key. Try this for line 10 above. Type 10 and press the ENTER key, then do a LIST. Only lines 5, 15, and 20 will remain as follows

```

5 PRINT "TI-99/4A"
10 PRINT 2+2
15 PRINT 10*10
20 PRINT "HELLO"
10
LIST
5 PRINT "TI-99/4A"
15 PRINT 10*10
20 PRINT "HELLO"

```

If you enter a line number with nothing after it, that line will be deleted.

How to Change Your Number

Your computer has a really neat command to help you tidy up program listings. In the previous example, we've got line numbers of 5, 15, and 20 left. Your programs will look neater and you'll have more room to insert lines if your program lines are in multiples of 10. So instead of lines 5, 15, and 20, it's more practical to have lines 10, 20, and 30. One way of

doing this would be for you to retype the entire program with line numbers as multiples of 10.

Happily, the designers of TI BASIC wanted to minimize your typing and included a great command which will automatically **renumber** lines. Another term for renumber is **resequence** and the command is called **RESEQUENCE**. You can abbreviate this command by just **RES** and your computer knows you want to resequence. Let's try this. Type in RES and press the ENTER key, and then do a LIST. You'll see

```
100 PRINT "TI-99/4A"
110 PRINT 10*10
120 PRINT "HELLO"
```

Notice that the RES command has resequenced lines 5, 15, and 20 in multiples of 10 starting with a line 100. If you want to resequence starting with a certain line number, just follow the RES command with this starting line number. For example, type RES 10, press the ENTER key, and do a LIST as shown below.

```
100 PRINT "TI-99/4A"
110 PRINT 10*10
120 PRINT "HELLO"
RES 10
LIST
10 PRINT "TI-99/4A"
20 PRINT 10*10
30 PRINT "HELLO"
```

As you can see, the program lines are now multiples of 10 starting from line 10. You can also use the RES command to resequence any way you want. For example, enter RES 500,35 then LIST, and you'll see

```
RES 500,35
LIST
500 PRINT "TI-99/4A"
535 PRINT 10*10
570 PRINT "HELLO"
```

The lines have now been renumbered starting at line 500 in increments of 35. So the general form of RES is

RES initial line number, increment

In this case, the increment is 35 since the line numbers go up by 35. Notice that the line numbers are not multiples of 35 in this example. The increment means the amount each line number is incremented or increased. Likewise, if you gave the command RES 9, the line numbers would be 9, 19, and 29 which are not multiples of 10. However the increment is 10.

If you give no arguments to RES, then the default values of 100 for the initial line and 10 for the increment are used. A **default** value is the value the computer assumes if you give no explicit instruction.

You can also use the default value for the initial line and specify an increment by just putting a comma before the increment. For example, enter the following

```
RES,5
LIST
100 PRINT "TI-99/4A"
105 PRINT 10*10
110 PRINT "HELLO"
```

By putting a comma before the increment, you are telling the computer to use the default value of 100 for the initial line and then use the increment of 5 that you give.

Now try to enter the following line

```
32768 PRINT 2-2
```

and you'll hear a beep and see the error message

```
* BAD LINE NUMBER
```

Now try

```
32767 PRINT 2-2
```

and the computer will accept it.

Line number 32767 is the highest line number your computer will accept. If you try to resequence past that, you'll see this error message. To show this, enter

```
RES 32700,100
```

and you'll see the BAD LINE NUMBER message. If any line of the program to be resequenced will have a line number exceeding 32767, then the RES command won't work.

A Little More LIST

You can list up to a certain line number by putting a minus or - before it. The "-" is a shifted key to the right of "P" that is also used for the minus sign. As an example of listing up to a line number, do a LIST -105 and you'll see

```
LIST -105
100 PRINT "TI-99/4A"
105 PRINT 10*10
```

will list all the program lines from the beginning of the program up to line 105.

Likewise

```
LIST 105-
105 PRINT 10*10
```

```
110 PRINT "HELLO"
32767 PRINT 2-2
```

will list all the lines from 105 to the end of the program.

Now do a LIST 105-110, and you'll see

```
LIST 105-110
105 PRINT 10*10
110 PRINT "HELLO"
```

Only lines 105 to 110 were printed.

In general, you may want to list out a long program to find a certain line. Just do a LIST to see the program scroll by or LIST starting at a line close to what you're looking for. Then hold down the FCTN and get ready to press the CLEAR key to stop the listing at any time.

What's NEW?

Suppose you have a program in memory and want to get rid of it. One way of doing this is turning power off. Another way is pressing the FCTN and QUIT keys. However, there's an easier way using the new command, **NEW**. Just type in

```
NEW
```

and press the ENTER key. The program in memory and any other data will be removed.

Enter the NEW command and the screen will clear. The message

```
TI BASIC READY
```

will appear at the lower left corner of the screen. If you try a LIST command now, the message

```
* CAN'T DO THAT
```

will appear because there is no program in memory.

Variables are also deleted. For example, enter in direct mode

```
A=3
PRINT A
```

You'll see a 3 appear. Now type in the NEW command and then

```
PRINT A
```

again. The value of 0 will be printed for A because the NEW command has erased the contents of the computer's RAM memory.

Easy Number

Another nice command you have is one that automatically generates line numbers. This is the **NUMBER** command, or just **NUM**. Type in NUM and press the ENTER key. You'll see a 100 appear below NUM and the

cursor will have moved 1 space past the line number of 100. Type in PRINT 1 and then press the ENTER key. Now a 110 will be printed and the cursor is again 1 space past the 110 line number.

You can continue this way, just entering commands and having line numbers automatically generated because you are in **Number Mode**. That's the name given to this mode of operation in which line numbers are automatically generated by the computer. But one question arises—how do you stop the automatic numbering? Easy. Just press the ENTER key without typing any command after the line number appears. The cursor will move to the next line and no line number will be there. Instead of pressing the ENTER key, you can also press the FCTN and "E" (up arrow) or FCTN and "X" (down arrow) keys. These appear to act just like the ENTER key.

The NUM command has options just like RES. For example, you can easily guess what NUM 10,20 should do. Go ahead and try it with the following example.

```
NUM 10,20
10 PRINT 1
30 PRINT 2
50 PRINT 3
70 PRINT 4
90 PRINT 5
110
```

where you press the ENTER key after 110 to end the automatic numbering. Just as the RES, the NUM command takes the initial line number as its first argument and the increment as its second argument. The default initial line number is 100 and default increment is 10, just as for RES.

NUM has another very interesting feature. Type in NUM 10,20 and you'll see line 10 appear on the screen with the cursor over the "P" of "PRINT". Use your right arrow key (FCTN and "D") to move the cursor to the right and change line 10 to

```
10 PRINT 10
```

then press the ENTER key. You'll see line 30 appear on the screen with the cursor over the "P" of "PRINT" just as in line 10. Edit this line to

```
30 PRINT 2+2
```

and press ENTER. Now line 50 appears. This time don't edit anything. Just press the ENTER key and line 70 appears. Press ENTER again and line 90 appears. Now edit line 90 to read

```
90 PRINT "HELLO"
```

press the ENTER key to store line 90, then press ENTER again to stop the NUM command.

From this example, you can see how easy it is to use NUM in stepping through your program lines for editing. If you want to edit a line, just use

the FCTN keys. If not, press the ENTER key to move on to the next. However, an important point to notice is that NUM never edited line 100. That's because NUM is editing in increments of 30 starting from line 10, and so 100 is never calculated.

Notice that you gave the command NUM 10,20 to edit this program. What do you suppose would happen if you had just done a NUM 10? Let's try it and see. The first line 10 appears for editing and you can edit that fine. But when you press the ENTER key again, line 20 appears instead of the line 30 that is in memory.

By giving only a NUM 10, the computer is just following orders and starting at line 10 in default increments of 10. While your program lines in memory are 10, 30, 50, 70, 90, and 100, the NUM command is going to take you 10, 20, 30, 40. . . If you press the ENTER key while line 20 is displayed and enter no command, you will just exit from Number Mode. However, if you enter some command, such as

```
20 PRINT 10*10
```

and then press ENTER, the computer will then show you line 30 for editing. If you want to insert line numbers between lines, then NUM 10 is correct. If you just want to edit existing lines, then you must use arguments for NUM that are suited for your line numbers.

Now let's go back to NUM mode by entering in NUM 30, 20. Line 30 appears with the cursor flashing over the "P" of "PRINT". Notice that you don't have to start from the beginning of the program to start editing in Number Mode. Just give the initial line number and increment. In fact, if you were only editing one line, you would not have to give the increment, no matter what it was.

While line 30 is displayed, press the FCTN and CLEAR keys. You'll see the cursor move down as you exit from Number Mode. The CLEAR key makes you exit from Number Mode immediately. Now do a NUM 30 to get back to line 30 and make the following change, but do not press the ENTER key.

```
30 PRINT 2*2
```

Instead, press FCTN and CLEAR after you change the "+" to an "*" in line 30. Now do a LIST and you'll see the original version of line 30

```
30 PRINT 2+2
```

come scrolling up the screen.

As you can see, using CLEAR immediately exits you from Number Mode and any changes you made to the line are not entered.

For our last example of Number Mode, do a NUM 30 again and then use the FCTN and ERASE keys to erase all the text of line 30. Press the ENTER key and you will be out of Number Mode. Now do a LIST and you will still see the original text of line 30 stored

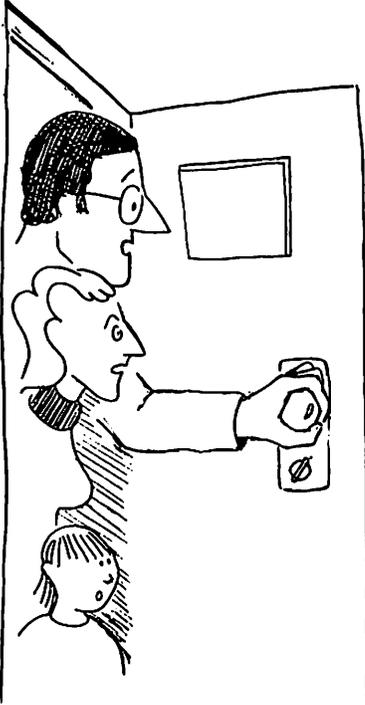
```
30 PRINT 2+2
```

The ERASE key deleted the command on a line in Number Mode. But if you then exit, the ERASE keys effects are not permanent, just like the CLEAR key.

Ups and Downs

There is another way to edit lines than using the NUM command. Instead the up and down arrow keys are used. For example, type in 30 (do not press ENTER) and then press the FCTN and "E" (up arrow key). Line 30 will appear on your screen ready for editing. If you want to edit only line 30, just press the ENTER key when you're done. However, if you want to edit the line before 30, press the FCTN and "E" key again. Your edited line 30 will be permanently changed, then the computer will show you line 10.

If you try to edit before line 10, you will just exit from this mode. Everytime you press FCTN and the up arrow key, you move up a line. Likewise, using FCTN and the down arrow ("X") key will move you down. You can switch back and forth from up to down arrows and vice versa anytime.



“I hear you’ve just gotten a new computer in the house.”

Chapter 4

Let's Get Variable

In this chapter, you'll learn a powerful concept in programming. Up to now we've looked at programs that did things like

```
10 PRINT 2+2
20 PRINT "HELLO"
```

so that you would get familiar with your computer. Now that you know how to enter and edit programs, you'll start learning to write practical programs that accomplish useful tasks.

A Shoebox Called MONEY

The concept that has made the computer so versatile is that of the variable. A **variable** is a named item that can store things. For example, enter the following program in which MONEY is the name of a variable

```
10 MONEY=100
20 PRINT MONEY
```

When you run this program, the computer tells you the value of MONEY.

MONEY is the name of a variable. The computer stores 100 in a certain part of its RAM and calls that by the name MONEY. You can imagine that the computer has a shoebox with the name MONEY. In line 10, you put a 100 into the shoebox. In line 20, you tell the computer to print the contents of the MONEY box. Every time you run this program, the computer puts 100 into MONEY, and then prints the contents of MONEY.

In line 10, you are assigning a value to a variable. Another way of writing this would be

```
10 LET MONEY=100
```

where the LET is optional. You don't have to type in **LET** and so most people don't.

Suppose you now edit line 10 to put 200 into MONEY. Run the program and you'll see the new value of MONEY printed.

You can also include text with your output. Change line 20 to

```
20 PRINT "MONEY=";MONEY
```

Now when you run the program, you'll get

```
MONEY= 200
```

If the money is in dollars, you can use this line

```
20 PRINT "MONEY=$";MONEY
```

and then you'll see after a run,

```
MONEY=$ 200
```

Adding an explanation of what is being printed is very important to whoever uses your program. While you may know what the program does, someone who did not write it may not. Even you may forget what is being output if you haven't used the program after a few weeks. Rather than having to read through the program listing and trying to understand it weeks or months later, you should always try to produce easily understandable output.

Bad MONEY

Notice that we printed the "\$" sign as part of the character string inside quotes. It's not possible to store a "\$" sign inside this variable because only a number can be stored. A variable that can store a number is called a **numeric variable**. You can include the normal part of a number such as a decimal point, or the number can be stored in exponential form. For example, change line 10 to

```
10 MONEY=200.53
```

and run the program. You'll see \$ 200.53 printed. For an exponential number, try

```
10 MONEY=2.0053E2
```

and run the program. Again, you'll see \$ 200.53 printed.

However, if you try change line 10 to

```
10 MONEY=$200.53
```

you'll hear a beep and see the error message

```
* BAD NAME
```

appear on the display because the "\$" sign cannot be stored in a numeric variable.

Variable names follow certain rules. For example, you can try all these versions of MONEY and the computer will accept them as legal names.

<u>Legal Name</u>	<u>Meaning</u>
@MONEY	at sign
\MONEY	backslash
[MONEY	left-bracket
]MONEY	right-bracket
__MONEY	Underscore or line

Note that the backslash is on the side of the "Z" key, not the "C". Variable names must begin with a letter or one of the symbols above.

Variable names can be up to fifteen characters long and contain only letters, numbers, the "@", and underscore "_". Following are some more legal numeric variable names

```
MONEYINSAVINGS
MONEY_IN_SAVING
MONEY@HOME
MONEY1
MONEY2
MONEY89654
M1_@_
```

Try entering these in line 10 and they will all be accepted. But illegal names like

```
MONEY IN SAVING
$MONEY
```

will be bad names and not accepted. Since spaces are not allowed in variable names, you can use the underscore to separate words in a variable name. Generally, programmers tend to use short variable names because it conserves computer memory and saves you typing. Every character of a variable name takes up 1 byte of memory. In a program with many statements or a lot of data, you may have to worry about conserving memory. However, the 16K bytes of your TI-99/4A give you a lot of storage.

The important thing to remember when you make up variable names is to make the names meaningful. If you are going to use a variable name for the money in your savings account, there are a number of ways to express it.

```
MONEYINSAVINGS
MONEY_IN_SAVING
MONEY\SAVINGS
MON_SAV
MONSAV
MSAVE
MONS
MS
M
```

Notice that it becomes harder for a reader to understand what the variable is storing as the name becomes more abbreviated. However, at least all these names begin with an M so there is some relation to money. A bad choice would be a variable name that had no relation to the purpose of the variable. For example, here are some bad choices for money:

```
X1
TIME
SNOW
WXY3
C@_ \
```

While all of these bad choices are legal variable names, they have no intrinsic or built in meaning to a person reading the program. You'll find it hard to understand what's going on in a program, even if you write it yourself, if good variable names are not used.

There are also certain words reserved for the computer that you can't use as variable names. For example, PRINT is a **reserved word**. A list of the reserved words is given in the Appendices. So you can't use PRINT as a variable name. However, you can use PRINT if it's part of a name. For example, PRINTA is legal for a numeric variable name. Likewise, PRINT\$ is a legal string variable name.

Stringing Along

Just as there are numeric variables, so too are there variables for storage of strings of characters. These are called string variables and are named like numeric variables except that a "\$" is always put at the end. For example, here are some legal string variable names

```
MONEY$
MONEY_IN_BANK$
M$
XYZ\@_ $
```

Enter and run this program to print a string variable

```
10 MONEY$="$200"
20 PRINT MONEY$
RUN
$200
```

Notice that there is no space between the "\$" and the "2" as there was when MONEY was a numeric variable. Now MONEY\$ contains a string of characters. These characters are just symbols to the computer and have no special meaning like numbers. For example, change line 10 to

```
10 MONEY$="CHEESE SANDWICH"
```

and run. Now you'll see the string "CHEESE SANDWICH" printed, not numerals like "200".

Numeric variables can be used in arithmetic operations while string variables cannot. For example, enter and run the following program as shown:

```
10 MONEY=200
20 PRINT MONEY;2*MONEY;SQR(M
```

```

ONEY);100-MONEY/5
RUN
200 400 14.14213562 60

```

As you can see, the value of the variable MONEY is used in all the arithmetic operations of line 20. Suppose you want to use a different value for MONEY? Just change line 10. For example, if MONEY = 100, let

```
10 MONEY=100
```

and run the program again. Now you'll see

```

RUN
100 200 10 80

```

You can now begin to appreciate how powerful is the concept of variables. Rather than your having to manually type in calculations for every input number, you just write a program in terms of variables. Do all your calculations in terms of variables and change only the initial line on which the variable's initial value is defined. In this case, you only had to change the value of the variable in line 10 since that's where MONEY is defined.

You can also combine strings with numbers in printing output. For example, enter the following program to print a sales receipt for a \$9.95 item.

```

10 PRICE=9.95
20 TAX_RATE=.05
30 PRINT "PRICE=$";PRICE;"TA
X=$";TAX_RATE*PRICE;"TOTAL=$
";PRICE+TAX_RATE*PRICE
RUN
PRICE=$ 9.95 TAX=$ .4975
TOTAL=$ 10.4475

** DONE **

```

In this program, line 10 defines the price as 9.95. Line 20 sets the tax rate as 5%. Line 30 prints the price, the tax and finally the total. Rather than performing the tax rate calculation twice in line 30, an alternate version of this program is

```

10 PRICE=9.95
20 TAX_RATE=.05
30 TAX=TAX_RATE*PRICE
40 PRINT "PRICE=$";PRICE;"TA
X=$";TAX;"TOTAL=$";PRICE+TAX

```

When you run this, you'll get the same answers as before. The important thing to note from this version is that we've reduced the amount of work the computer has to do. Instead of calculating TAX_RATE*PRICE twice in the PRINT line, this calculation is only done once in line 30 now. Multiplications, divisions and other math operations take more time for

the computer to execute than simple additions and subtractions. While it does not matter much in this simple example, more complex programs may involve thousands of calculations. People think of computers as fast until they have to wait for the computer to calculate an answer to a problem. Then you'll start thinking how slow they are.

However, nothing is free. Although we've gone down from two to one tax rate calculations, we've also added another line to the program. The computer takes some time to execute this additional line compared to the first three-line program. In this particular case, it takes about the same time for both versions of the program.

Give Me Cents

In working with programs dealing with money, it's best to print only dollars and cents. Notice that as the result of calculations, the tax and total are printed with four digits after the decimal point. Let's use the INT function to round off results to two decimal places and then RUN.

```

10 PRICE=9.95
20 TAX_RATE=.05
30 TAX=INT(100*TAX_RATE*PRIC
E+.5)/100
40 PRINT "PRICE=$";PRICE;"TA
X=$";TAX;"TOTAL=$";PRICE+TAX
RUN
PRICE=$ 9.95 TAX=$ .5
TOTAL=$ 10.45

** DONE **

```

The exact tax of .4975 is rounded off to .50. However, the computer does not print the trailing 0, so only a .5 is printed. As another example, change line 10 to

```
10 PRICE=12.95
```

Now when you run, you'll see

```

PRICE=$ 12.95 TAX=$ .65
TOTAL=$ 13.6

```

Notice that the tax is indeed rounded off to two decimal places. To find out its exact value, type in this direct command to multiply the price times rate and press the Enter key.

```
PRINT 12.95*.05
```

You'll see .6475 printed as the exact tax. This .6475 is rounded off to .65 by line 30. You can also see that the trailing 0 of the TOTAL is now left off in printing output. Instead of \$ 13.60, the computer leaves off the trailing 0 in printing 13.60.

Also try as a direct command

```
PRINT PRICE*TAX_RATE
```

You'll see the same result of .6475 printed.

The reason this occurs is that variables retain their last value when the program ends or is interrupted. You can always find the latest value of a variable by just doing a PRINT or even calculations as shown here.

You can even change the value of a variable by a direct command. For example, do a

```
PRICE=50
PRINT 2*PRICE
100
```

and you'll see a 100 printed since that is the value of 2*PRICE.

Besides numeric variables, you can also print string variables. For example, enter the following

```
10 PRICE=9.95
20 TAX_RATE=.05
30 TAX=INT(100*TAX_RATE*PRIC
E+.5)/100
40 TOTAL=PRICE+TAX
50 PRIS="PRICE=$"
60 TAX$="TAX=$"
70 TOT$="TOTAL=$"
80 PRINT PRIS;PRICE;TAX$;TAX
;TOT$;TOTAL
```

When you run this you'll see the same output as the previous program of 9.95 for the price, .5 for the tax, and 10.45 for the total. Lines 50-70 define string variables which are printed along with numeric variables in line 80. Sometimes, you may use the same string of characters several places in the program. You'll conserve memory and do less typing if you define string variables instead of directly printing the same strings in several places.

Another point to note about this program is that all calculations are now stored in variables. If you will need the results of the calculations, such as tax and total, for later work, you'll want to store the results in variables.

How Much Can You Store in a String?

You can store up to 255 characters in a string variable. Any characters over 255 will be discarded. The resulting string is said to be **truncated** if it is limited to a certain number of characters. Truncation means only allowing a certain number of characters.

One question you may have is how do you get 255 characters in a string since you can only input four lines of characters? At 28 characters per line, that's 28X4 = 112 characters. And you can't even put 112 into a string since some space must be reserved for the variable name, line

number, quotes and "=" sign. The answer is shown in the following program. Before entering it, do a NEW command to clear the price program out of memory by typing NEW and then pressing the ENTER key. To enter all the "A"'s, hold down the "A" key until the computer beeps, then replace the last "A" with a quote, ", and press ENTER.

```

10 A$="AAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAA"
20 A$=A$&A$&A$
30 PRINT A$

```

The way to make big strings is by concatenation. In line 20, we concatenate the strings. Each A\$ has 104 characters in it, so the A\$ of line 20 should have $104 * 3 = 312$ A's

When you run this program, you'll see there are only 255 A's. The rest have been ignored by the computer. The final A\$ from line 20 thus has only 255 A's in it.

Chapter 5

Giving Input

Until now, the only way you've seen how to change input is to change the program. In this chapter, you'll learn a much more convenient and easy method to input data. We'll also discuss more efficient ways of storing data.

Easy Input

Your computer has a simple command which lets you input data without having to change a program line. As you might have expected, this is the **INPUT** command. Enter and run the program below. When you see the "?" mark and hear a beep, type a "2" as shown and then press the ENTER key.

```
10 INPUT N1
20 PRINT N1
RUN
? 2
  2
** DONE **
```

When the computer executes the INPUT statement of line 10, it sounds a beep and prints a question mark. This question mark is to prompt you to enter input. The input you give is assigned to the numeric variable N1 in line 10. Then the computer executes line 20 and prints the value of N1, in this case, 2. Run the program for each of the following input values and you'll see them printed out each time: 25, 100.865, 10000000000.

Now run the program again for an input with one more zero, of 100000000000, and you'll see the answer printed in exponential notation of 1.E+10. Also, try inputting a number with commas to separate digits. For example, enter 1,000. The message

```
* WARNING:
  INPUT ERROR IN 10
  TRY AGAIN:
```

will appear because a comma cannot be part of a number in BASIC. However, you'll get this same error message if you try to input an alphabetic character or other printable symbol not part of a number. For example, try

to enter an "A" and you'll see this same warning. You can enter numbers in exponential notation such as 1.23E3, -7.69E-8. These will be printed as 1230 and -.00000000769 respectively.

Now let's add a few statements to this program so that it does something a little more useful than print the input number. Add these lines and then run as shown to add 2 + 10.

```

30 INPUT N2
40 PRINT N2
50 PRINT "SUM=";N1+N2
RUN
? 2
  2
? 10
 10
SUM= 12

** DONE **

```

Line 30 prompts you for the second number, N2. Line 40 prints N2 while line 50 prints the sum of N1 and N2. Every time you run this program, you'll be asked to input two numbers and then the program will print the result. You can easily see how this program can be expanded to print any math operation.

When you write a program to ask for input, it's always a good idea to prompt the user as to what is needed for input. For example, you could add lines like

```

5 PRINT "FIRST NUMBER="
25 PRINT "SECOND NUMBER="

```

When you run the program for inputs of 2 and 10 you'll see

```

RUN
FIRST NUMBER=
? 2
  2
SECOND NUMBER=
? 10
 10
SUM= 12

** DONE **

```

You can get a different appearance by adding a semicolon to the ends of lines 5 and 25. When you run, you'll see

```

RUN
FIRST NUMBER=? 2
  2
SECOND NUMBER=? 10

```

```
10
SUM= 12
```

```
** DONE **
```

Notice that by adding the semicolon at the end of lines 5 and 25, the following question marks from the input statements of lines 10 and 30 are printed immediately following the equal signs.

However, there's an easier way of writing this program. You can include the text with the INPUT rather than on a separate line. To show this, first enter a NEW command to delete the program from memory. Then enter and run as follows.

```
10 INPUT "FIRST NUMBER=":N1
20 INPUT "SECOND NUMBER=":N2
30 PRINT "SUM=";N1+N2
RUN
FIRST NUMBER=2
SECOND NUMBER=10
SUM= 12
```

```
** DONE **
```

In lines 10 and 20, the input prompt is enclosed in quotes, then the name of the variable follows a colon. Also, notice that when run, a question mark does not automatically appear when an INPUT statement is executed. The question mark only appears if there is no input prompt. If you want a question mark, you'll have to include it as part of the input prompt. For example,

```
10 INPUT "FIRST NUMBER=?":N1
```

Another way of writing this program that's even shorter is shown following. First enter a NEW command. Then enter and run the following program as shown.

```
10 INPUT "FIRST,SECOND NUMBE
R=":N1,N2
20 PRINT "SUM=";N1+N2
RUN
FIRST,SECOND NUMBER=2,10
SUM= 12
```

```
** DONE **
```

Notice that although only one input prompt is allowed on an INPUT, you can include more variables. This variable list has the variables separated by commas. When you input the numbers during the run, just separate the input values by commas also. If you try inputting only one value and pressing ENTER, you'll get the error message

```
* WARNING
  INPUT ERROR IN 10
  TRY AGAIN:
```

Likewise, if you try separating the input values by a ";" or other non-comma symbol, you'll get this error message.

If you're ever trying to give a program input variables and it won't accept them, you'll be stuck on the INPUT statement unless you press FCTN and CLEAR. This halts the program and you can then try to find out what's wrong by looking at the INPUT statement.

You can also input strings by themselves or with numeric variables. First, enter a NEW to delete the current program. Then enter the following and run

```
10 INPUT "FIRST, LAST NAME ":
  FIRST$, LAST$
20 PRINT FIRST$; " "; LAST$
RUN
FIRST, LAST NAME JOHN, SMITH
JOHN SMITH
```

Line 10 asks for the first and last name and assigns the input strings to variables FIRST\$ and LAST\$. Notice the space after the "NAME" in line 10. Without this space, the "J" of "JOHN" would be printed immediately after the "E" of NAME. You'd see

```
FIRST, LAST NAME JOHN SMITH
```

An alternative would be

```
10 INPUT "FIRST, LAST NAME?":
  FIRST$, LAST$
```

However, a space after the question mark is still helpful when you read the input prompt.

You can also surround the input strings with quotes and you'll see the same output. For example,

```
RUN
FIRST, LAST NAME "JOHN", "SMITH"
JOHN SMITH

** DONE **
```

The quotes are optional except when you want to input quotes, commas, leading spaces, or trailing spaces. For example, try these

```
RUN
FIRST, SECOND NAME "JOHN, ", SMITH
JOHN, SMITH

** DONE **
```

```

RUN
FIRST,SECOND NAME "  JOHN","S
MITH "
      JOHN SMITH

```

In the first example above, a comma is included after JOHN. In the second example, three leading spaces are input before "JOHN" and two trailing spaces are input after "SMITH"

Likewise you can include quotes

```

RUN
FIRST,LAST NAME """"JOHN"""" ,S
MITH
"JOHN" SMITH

```

**** DONE ****

Here's an example of a program which combines string and numeric input. First enter a NEW command, and then enter and run the following program.

```

10 P$="FIRST, LAST NAME, SALAR
Y="
20 INPUT P$:FIRST$,LAST$,SAL
30 PRINT : "EMPLOYEE":
40 PRINT LAST$;" ";FIRST$
50 PRINT "SALARY=";SAL
RUN
FIRST, LAST NAME, SALARY=JOHN,
SMITH,1000

```

```

EMPLOYEE
SMITH,JOHN
SALARY= 1000

```

**** DONE ****

Line 10 defines a string variable which is used as the input prompt in line 20. Line 20 inputs the first name, last name, and salary. Line 30 prints a blank line before EMPLOYEE to make it easier for people to read the program's output. Lines 40 and 50 then print the employee name and salary.

From this example, you can see how the same type of program can keep track of inventory, recipes, record collections and any other items. In a store, this type of program could record the item, price and then total cost after taxes.

For example. enter the following new program to print an item name and total price for a sales tax of 5%.

```

10 TAX=.05
20 INPUT "ITEM,PRICE=?":ITEM
$,PRI
30 PRINT ITEMS$;" $";PRI+TAX
*PRI

```

Now run this as follows

```

RUN
ITEM,PRICE=?BREAD,.40
BREAD $ .42
** DONE **

```

Now run the program again for flour at 1.45, and you'll get \$1.5225 as the final price plus tax. Just as in the sales tax example of the previous section, let's round the result off to two decimal places by replacing line 30 with

```

30 PRINT ITEMS$;" $";INT(100
*(PRI+TAX*PRI)+.5)/100

```

When you run this for flour at 1.45, the answer is rounded off to the nearest cent and you'll see \$1.52.

Let's Make Money

Now that you understand INPUT, let's look at a few practical programs involving money. For example, suppose you put \$1000 into a bank which pays 12% interest. How much will you make in three years?

Let's assume the bank pays interest only once a year. So after the first year, you have

$$1000 + 1000 * .12 = 1120$$

Try the above calculation in direct mode with a PRINT command to check the calculations. The amount you invested was the **principal** of \$1000. Your interest after one year was the principal times the interest rate of 12%. So the interest you earned was

$$1000 * .12 = 120$$

where the percent interest is converted to a decimal, .12. The interest of 120 is added to your original principal of 1000 to give your new principal of $1000 + 120 = 1120$.

What about the second year? Now your principal is \$1120 and so

$$1120 + 1120 * .12 = 1254.4$$

For the third year, you have

$$1254.4 + 1254.4 * .12 = 1404.928$$

Again, you might check these calculations with a PRINT command.

By now you can see a pattern forming. In fact, there is an easy formula to calculate this for three years. Try the following

```
PRINT 1000*(1+.12)*(1+.12)*(
1+.12)
```

and you'll get 1404.928 just as before.

An easier way of writing this formula is with exponents. Try

```
PRINT 1000*(1+.12)^3
```

The exponent of 3 is used because the same factor of $1 + .12$ appears three times in the calculations.

The formula above is very convenient in calculating the total amount you've earned. Suppose you want to know how much you'll make in 20 years. Just enter

```
PRINT 1000*(1+.12)^20
```

and you'll see

```
9646.293093
```

Of course, just as with the other examples of dollars and cents calculations, you may wish to round the results off to two decimal places.

Using this formula, it's easy to write a program to calculate the amount you can earn. Enter

```
10 INPUT "PRINCIPAL,INTEREST
RATE (%),YEARS=":PR,IN,YE
20 TOT=PR*(1+IN/100)^YE
30 PRINT "TOTAL=$";INT(100*T
OT+.5)/100
```

Line 10 asks you to input the principal, interest rate as a percent, and number of years. These values are stored in the variables PR, IN, and YE respectively. Line 20 does the calculation and stores the total amount you've earned in the variable TOT. Notice that since the interest rate was input as a whole number, it must be divided by 100 to convert it to a decimal. Line 30 then prints the result after rounding the total amount to the nearest cent.

Now run this program as shown for a principal of 1000, at 12% interest for 20 years:

```
RUN
PRINCIPAL,INTEREST RATE (%),
YEARS=1000,12,20
TOTAL=$ 9646.29
```

```
** DONE **
```

You may also wish to calculate the total amount you've earned by subtracting your original principal. So just add this line

```
40 PRINT "AMOUNT EARNED=";T
OT-PR
```

and run it as shown.

```
RUN
PRINCIPAL,INTEREST RATE (%),
YEARS=1000,12,20
TOTAL=$ 9646.29
AMOUNT EARNED=$ 8646.293093
```

```
** DONE **
```

Notice that the amount earned was not rounded off to the nearest cent. Try changing the program to do this. You could either define a new variable for the amount earned and round it off or just round off TOT-PRINCIPAL in line 40 like this

```
40 PRINT "AMOUNT EARNED=";I
NT(100*(TOT-PR)+.5)/100
```

Now you'll see the amount earned of \$8646.29, which is rounded to the nearest cent.

How to Make More Money

Up to this point, we've assumed the bank pays interest only once a year. This was done in order to simplify the program. However, banks normally pay interest on a monthly basis. In fact, some banks even pay daily interest. Paying more often than once a year is called **compound interest** because your interest builds up more than once a year. The time between interest payments is called the **period**. Let's see how to make our program handle interest payments more often than once a year. The way to do this is to divide the interest rate by the number of payments per year. For example, if the yearly interest rate is 12%, then the monthly interest rate is 1%. This monthly interest rate is the interest rate you use in computing the compound interest.

Change the program by changing lines 10 and 20 as follows.

```
10 INPUT "PRINCIPAL,INTEREST
RATE (%),YEARS,PERIODS/YEAR
=":PR,IN,YE,PER
20 TOT=PR*(1+IN/100/PER)^(YE
*PER)
```

Line 10 now asks how many periods there are per year. Line 20 calculates the interest per period by

```
IN/100/PER
```

For example, if IN=12, then the monthly interest is .01. Also, the total number of periods must be used in the exponent, so this is

YE*PER

For example, if there are 12 periods per year and 5 years, then there are 60 monthly payments. Likewise, with daily interest, there are 365.25 periods per year (allowing 1 day for a leap year every four years on the average). Of course, your bank only pays the extra day's money on a leap year so you'll have to wait four years to get the extra day's interest.

Let's run the program for a principal of 1000, at 12% interest per year, for 5 years at 12 periods per year. The result is \$1816.7 for the total. Now try it for a daily interest with 365.25 periods per year. Now the total is \$1821.94. As you can see, the daily rate doesn't earn you that much more than the monthly rate. Just for fun, see how much money you'd earn for an hourly interest rate, and then for a rate compounded every second. You'll be amazed at how little more you earn. Although it sounds great to earn hourly or secondly interest, your computer can really help you figure out the true facts and make the best decision. For example, what's better: 12% interest compounded monthly or 10% interest compounded daily? You can really profit by using your computer to analyze investments like this.

How's Your IRA?

As another practical application, let's see how much money you can make with an **Individual Retirement Account (IRA)**. With an IRA, an employed person can contribute up to \$2000 a year and not pay any taxes until the money is withdrawn. On a regular savings account, you do pay taxes on the interest. So the formulas we developed for compound interest work for an IRA if you make a single contribution. However, with an IRA, you can make additional contributions every year and so your money grows much faster.

The following program is a modification of the compound interest program. You may wish to save the compound interest program before making the following changes for an IRA. If you have a suitable cassette recorder hooked up, just type in the **SAVE** command

SAVE CS1

press ENTER, and follow the instructions given by the computer for **saving programs**. If there is no program in memory, the error message **"* CAN'T DO THAT"** will appear. To **load** a program back into your computer from cassette, enter the **OLD** command

OLD CS1

press ENTER, and follow the instructions from your computer. The following IRA program computes how much you'll earn under an IRA if you invest the same principal every year. Also, note that the program lines are longer than the 28 columns shown on your screen. Just enter the lines as you normally do. These lines are shown longer because it's easier for you

to understand what is being entered. From now on, we'll use these longer lines.

```

10 INPUT "PRINCIPAL, INTEREST RATE (%), YEARS, PERIODS/
YEAR=":PR, IN, YE, PER
15 IN=IN/100/PER
20 TOT=PR*((1+IN)^(YE*PER+1)-(1+IN)^PER)/IN
30 PRINT "TOTAL=$";INT(100*TOT+.5)/100
40 PRINT "AMOUNT EARNED=$";INT(100*(TOT-PR*YE)+.5)/100

```

Line 10 asks the information needed by the program. Line 15 computes the decimal interest per period and assigns it to the variable IN. This variable was defined for convenience. If you look at line 20, which computes the total, you'll see IN is used three times. So using IN saves your typing in the expression

IN/100/PER

three times in line 20. Line 30 prints the total while line 40 shows how much you actually earned. In line 40, notice that the principal, PR, is multiplied by the number of years. This is because you've invested the same amount every year. For example, if you invest \$2000 for 20 years, then you've put in \$40,000.

Now run this program for a principal of 2000, 12% interest, 20 years and 12 periods per year. You'll see a total of \$1974930.83 and an amount earned of \$1934930.83. So your \$40,000 investment over 20 years gets you \$1,934,930.83.

You should save the programs that you enter. In many cases we'll use these as building blocks to add more features as we discuss new commands. Rather than overwhelming you with a long, complete program, we'll build up the program in easy to understand stages. For example, the Database and R-Walk, programs discussed in later chapters are built up in stages. As we cover new commands, you'll see how to enhance these programs.

Can You Afford It?

If you've ever wondered whether you could afford payments on a loan, then this program can help. For example, suppose you want to buy a car and need an \$8000 loan. What will your monthly payments be for different interest rates and periods/year? Enter the following program or just modify the IRA program using the editing keys so that it looks like this. As before, note that the statements below are written out with more than 28 columns so that you can read them more clearly.

```

10 INPUT "PRINCIPAL, INTEREST RATE (%), YEARS, PERIODS/
YEAR=":PR, IN, YE, PER
15 IN=IN/100/PER
20 INC=(1+IN)^(YE*PER)
25 PAY=PR*IN*INC/(INC-1)
30 PRINT "PAYMENT=$";INT(100*PAY+.5)/100

```

and delete line 40. To delete a line, just type in the line number and press the ENTER key.

Now run this program for the example below.

```
RUN
PRINCIPAL,INTEREST RATE (%),
YEARS,PERIODS/YEAR=8000,12,3
,12
PAYMENT=$ 265.71
```

As you can see, this program easily calculates the monthly payment for you as \$265.71. By trying different interest rates, you can determine what interest rate you need for a payment you can afford.

In this program, line 20 defines a variable for convenience called INC. This stores the interest calculated for the total number of periods, $YE * PER$. Line 25 calculates the payment while line 30 prints the result. You may also wish to simplify your input of data. Since car loans are usually figured on a monthly basis, you can eliminate that input request and always set $PER = 12$.

Your computer can also help you get the best bargain in a loan by also revealing how much the loan will ultimately cost you. Just because your monthly payments on one loan are lower than another doesn't mean that loan is less. If the payments are spread out over a longer time, you may pay more. Let's change this program to reveal the total amount we'll pay and the total interest. Add these lines.

```
40 PRINT "TOTAL COST=$";INT(100*YE*PER*PAY+.5)/100
50 PRINT "INTEREST=$";INT(100*(YE*PER*PAY-PR)+.5)/100
```

Line 40 calculates the total amount you pay by multiplying the total periods, $YE * PER$, by your monthly payment, PAY . This term

```
YE*PER*PAY
```

is the total amount you've paid.

Line 50 calculates the interest on your loan by subtracting the principal from the total amount you've paid. The difference is the interest on the loan.

Run this program for the same example of 8000 at 12% for 3 years at 12 periods per year. In addition to the monthly payment of \$265.71, you'll also see

```
TOTAL COST=$ 9565.72
INTEREST=$ 1565.72
```

which shows how much you'll ultimately pay and the interest charge.

Now let's comparison shop. Suppose you can get a 10% loan for 5 years. Are you really better off? Let's try it and see as shown below.

```
RUN
PRINCIPAL,INTEREST RATE (%),
YEARS,PERIODS/YEAR=8000,10,5
,12
PAYMENT=$ 169.98
TOTAL COST=$ 10198.58
INTEREST=$ 2198.58
```

So although your monthly payments are almost \$100 less with the 10% loan than the 12% loan, you must pay back \$632.86 more in interest.

Chapter 6

Going Automatic

One of the biggest advantages of a computer over a simple calculator is that the computer can execute statements automatically. That is, the computer can make decisions and go to certain statements without your continual supervision. In this chapter, you'll learn how to make your computer operate automatically.

Go To It

In the programs you've seen, the program did its thing and then ended. It's usually more convenient for the program to ask you for more input so that you don't have to enter a RUN each time. You can easily do this with the **GOTO** statement. Just use GOTO followed by the line number you want the computer to go to.

To see how GOTO works, first delete any existing program in memory with a NEW command. Then enter this simple program.

```
10 INPUT "FIRST,SECOND NUMBER=?":N1,N2
20 PRINT "SUM=";N1+N2
30 GOTO 10
```

Line 10 asks you to enter two numbers which are assigned to the variables N1 and N2, respectively. Line 20 prints the sum. Line 30 tells the computer to go to line 10 because 10 is the argument of GOTO. Now run this program for the example below, where you input 5 as the first number and 10 as the second.

```
RUN
FIRST,SECOND NUMBER=?5,10
SUM= 15
FIRST,SECOND NUMBER=?
```

When you run the above example, the computer will ask you for input from line 10 and print the sum in line 20. Next the computer executes line 30 which directs it to go back to line 10. When the computer goes back to line 10, it asks you for input again.

You can now enter two more numbers and the computer will print their sum and then go back to line 10. This process of the computer asking for input and printing the sum could go on forever. The computer is running the program automatically. You don't have to enter RUN every time.

At this point, you may also be wondering how you can stop the program. There are several ways. First, you could turn off the power, or press the FCTN and QUIT keys. However, you would lose the program since the contents of the computer's memory are erased under these conditions.

Let's Continue

Another way to stop the program is to use FCTN and CLEAR. Try this and you'll see the message

```
* BREAKPOINT AT 10
```

appear on your screen. This **BREAKPOINT** message means that the program has stopped at line 10. If you'd like to continue the program now, just type in the word **CONTINUE** or its abbreviation **CON**, and press the ENTER key. The computer will continue from the breakpoint line 10.

There's a big difference between starting the program with a CONTINUE rather than the RUN. Stop the program with a FCTN and CLEAR and then enter the command

```
PRINT N1;N2
```

If you haven't input any more numbers from the original 5 and 10 you'll see the 5 and 10 printed. If you have input other numbers, the latest values for N1 and N2 will be printed.

Now RUN the program again, and when the computer asks for the first and second number, press the FCTN and CLEAR keys. Now do a PRINT N1;N2 again. This time they will both be 0 since the RUN command sets the value of all numeric variables to 0, and sets all string variables to no characters—the **null string**. Just as numeric variables are initially set to zero, all string variables are set to have no characters. For example, enter the direct command

```
PRINT "*" ; X$ ; "*"
```

and you'll see "***". The left asterisk is printed, then the null string of X\$, i.e., no characters, and finally the right asterisk is printed. The asterisks are used here to **delimit** the string for X\$. The term delimit means to define the limits. Without the asterisks, you might argue that X\$ contained a blank. However, a blank is a character and so you would have seen "* *", i.e., a blank space between the asterisks.

You can continue from a program that's been halted with CLEAR if you make no changes in the program. Try this. First input 5 and 10 again and then halt the program with FCTN and CLEAR when the computer requests input again. Now add the line

```
25 PRINT
```

and then try a CON. The message

```
* CAN'T CONTINUE
```

will appear because the program has been changed.

You'll find it convenient in debugging programs to use CLEAR. The term **debugging** means to find and to fix the errors in a program. Another way of saying this is that a program has bugs or problems in it. So when you debug, you get the bugs out. By interrupting a program with CLEAR, you may check the values of suspicious variables and compare them to what you think they should be. If the values of the variables are different, then you can try to figure out what the problem is. By checking the values and continuing the program until bad values occur, you will eventually find the bad program lines. Another way of checking is to add lines to print the value of suspicious variables. After the program is debugged, just delete these debugging print lines. In a later chapter we'll discuss other powerful techniques for debugging that your computer has available.

Caught in a Loop

When you tell the computer to do a GOTO, you should be sure the line number makes logical sense. For our example of adding two numbers, the GOTO 10 does make sense. However, suppose you change line 30 to

```
30 GOTO 25
```

and run the program for inputs of 5 and 10. You'll see the sum of 15 printed and then everything goes scrolling up the screen. In a few seconds, the screen is entirely blank. What's happened?

In this case, line 30 tells the computer to go to line 25 and print a blank line. The computer executes line 25 to print a blank line, then goes to line 30 which tells it to go back to line 25 and print a blank line. The computer just keeps on executing lines 25 and 30. When a computer keeps executing the same lines over and over again with no way of stopping, the computer is said to be caught in a **loop**. The computer keeps looping back through the same lines all over again. To stop, do a FCTN and CLEAR.

The computer doesn't care that it keeps executing the same lines over and over again. It's perfectly happy doing exactly what you tell it even if the instructions make no logical sense. You must supply the intelligence to make the computer accomplish a useful purpose.

The GOTO is very convenient in many cases. You may wish to put it at the end of the money programs we've discussed. Also, it helps you to read the output if you put a PRINT line or two before the GOTO.

Getting Control

The GOTO is an example of an **unconditional jump** or **branch** to another line. The term unconditional jump means that whenever the computer comes to a line with a GOTO, it always goes to the line number specified by the GOTO. However, there is a variation of the GOTO that provides a conditional jump. For example, enter the following program.

```

10 INPUT "FIRST,SECOND NUMBER=?":N1,N2
20 PRINT "SUM=";N1+N2
30 INPUT "ENTER 1 TO CONTINUE;2 TO END ":D
40 ON D GOTO 10,50
50 END

```

Now run this program for $N1 = 5$ and $N2 = 10$. After the sum is printed, enter a 1 as input for D in line 30. You'll then see the program ask you for the first and second numbers again. Enter any two numbers and you'll see their sum printed. Now when the computer asks whether you want to end the program, input a 2, and you'll see the program end. For example

```

RUN
FIRST,SECOND NUMBER=?5,10
SUM= 15
ENTER 1 TO CONTINUE;2 TO END
1
FIRST,SECOND NUMBER=?2,2
4
SUM= 4
ENTER 1 TO CONTINUE;2 TO END
2
** DONE **

```

Depending on the value of D, line 40 controls which line the computer will execute next. The command used in line 40 is the ON GOTO. Following the **ON** can be any numeric expression. If the numeric expression is 1, the computer will execute the first line in the list after the GOTO. So when $D=1$, the computer executes line 10. If $D=2$, the computer executes the second line number after the GOTO. In this case, the computer goes to line 50 and the program ends.

An **END** statement always ends the program execution. The END can be located anywhere in the program. That is, the END does not have to be the highest line number. Another statement that will end execution is **STOP**. Try substituting STOP for END and you'll see the program act the same as it did for END.

What happens if you don't enter a 1 or 2? Let's see. Run the program again for any two numbers you want. Then when the computer asks you for a 1 or 2, enter a 1.4. You'll see the computer just continue with the program. Enter two more numbers for the sum. Now when the computer asks for a 1 or 2, enter a 1.5. You'll see the program end. These results occur because the value of the numeric expression is rounded off by the ON GOTO to the nearest integer. So 1.4 is rounded off to 1 and the program goes to line 10. The 1.5 is rounded off to 2 and the program goes to line 50 and ends.

The numeric expression can be anything which yields a number. For example, you could use a constant such as 1 as in

```
40 ON 1 GOTO 10,50
```

to make the program always go to line 10. However, this is the same as an unconditional jump like

```
40 GOTO 10
```

and so it's more efficient to use a GOTO. You can use a constant numeric expression like

```
40 ON 1+1 GOTO 10,50
```

and the computer will always end the program since $1 + 1 = 2$ and so the ON 2 GOTO 10,50 always goes to line 50 to end. It's more efficient to use a GOTO directly in this case. Of course, you can use a numeric expression involving variables, such as

```
40 ON 2*D+1 GOTO 10,50
```

But be careful if you input a value for D that causes $2*D + 1$ to exceed 2 or be less than 1. For example, with $D=1$, you'll get $2*1 + 1 = 3$ and you'll see the error message

```
* BAD VALUE IN 40
```

Let's Be Logical

Besides the arithmetic operators, your computer also has some **logical operators**. The logical operators test whether a **relation**, also called a relationship, is true or false. For example, try this direct command

```
PRINT 1=1
```

and you'll see a

```
-1
```

printed. Also try the following PRINT commands and you'll see the results shown below each PRINT.

	<u>Meaning</u>
PRINT 2=2 -1	2 equals 2
PRINT 35.9>18 -1	3.59 is greater than 18
PRINT -3<-2 -1	-3 is less than -2
PRINT 2+3>8/4 -1	5 is greater than 4
PRINT 4<>3 -1	4 is unequal to 3

As you can see, whenever the relationship is true, the computer prints a value of -1. This value of -1 is the computer's way of saying that the logical relationship is true. The computer can test these relational operators.

<u>Operator</u>	<u>Meaning</u>
=	equal
>	greater than
<	less than
<>	unequal
<=	less than or equal to
>=	greater than or equal to

You can also test variables. For example, first do the direct commands

```
X=3
Y=4
```

and then the PRINT

```
PRINT X<>Y
-1

PRINT X<Y
-1

PRINT X+1=Y
-1
```

The real value of relational tests lies in programs where the values of variables are being tested.

Besides numeric variables and numbers, you can also test string variables and characters. For example

	<u>Meaning</u>
PRINT "A"="A" -1	the character "A" equals the character "A"
PRINT "A"<>"B" -1	the character "A" is unequal to the character "B"

To test string variables, first define

```
X$= "TOM"
Y$= "DICK"
```

and then

```
PRINT X$=X$
-1
```

since "TOM" does equal "TOM." Also

```
PRINT X$<>Y$
-1
```

is true since "TOM" is not the same as "DICK."

So far we've looked at expressions which are all true. Now let's look at some false relationships. Try the following.

```
PRINT 1=3
Ø
PRINT X=Y
Ø
PRINT 2*X<Y+1
Ø
PRINT "A">"B"
Ø
```

Notice that in the last example above, the computer says that the character "A" is not greater than the character "B". Try

```
PRINT "B">"A"
-1
```

and you'll see that the computer believes "B" is greater than "A." The reason this occurs is because the computer actually compares a code number for each letter. These codes are defined by the American Standard Code for Information Interchange (ASCII). The codes for the printable symbols of your BASIC are in the Appendix of ASCII Character Codes. For example, the code for "A" is 65, and the code for "B" is 66. In testing "A"="B" the computer really checks whether 65=66 is true or false.

Now try

```
PRINT "AA">"B"
Ø
```

and you'll see the result of 0 which means false. In comparing strings of unequal length, the computer compares each character of the shorter string against the longer and stops when there are no more characters. So "AA," "AB," etc., are all considered less than "B."

To test string variables, try

```
PRINT X$=Y$
Ø
```

and the result is 0 since "TOM" does not equal "DICK."

You can use the relational operators to shorten the program which prints the sum of numbers. For example, try this

```
1Ø INPUT "FIRST,SECOND NUMBER=?":N1,N2
2Ø PRINT "SUM=";N1+N2
3Ø ON 1-(N1=Ø) GOTO 1Ø,4Ø
4Ø END
```

and run for 5,10. Then when the program asks for input again, enter 0,0 and you'll see the program end after printing a sum of 0.

In this program, line 30 checks if the first number input is a 0. If so, then $N1 = 0$ is true and the computer returns the value -1 for it. Then the computer evaluates the rest of the relational expression in line 30 as $1 - (-1) = 2$ and so goes to the second statement after the GOTO, which is line 40. However, if $N1$ is not 0, then $N1 = 0$ returns a value of 0 and the relational expression is $1 - 0 = 1$. Now the computer goes to the first line number after the GOTO in line 40, which is line 10.

When you read a program using relational operators, don't be misled by appearances. Remember that line 20 in the following program

```
10 Y=0
20 X=Y=0
30 PRINT X;Y
```

does not set X to 0. Rather, the relation $Y = 0$ is evaluated as true by the computer and so X is set to -1 in line 20. Try this program and see.

If This Is True

Your computer has a very general test for relationships called the **IF** test. Enter the following program and run it for 5,10 and then 0,0. You'll notice the program ends when the 0,0 is input.

```
10 INPUT "FIRST,SECOND NUMBER=?":N1,N2
20 IF N1=0 THEN 50
30 PRINT "SUM=";N1+N2
40 GOTO 10
50 END
```

Line 20 is the IF test. The relationship following the IF is tested. If the relationship is true, then the computer executes the line number which follows the THEN. In this case, when $N1 = 0$ is true, the computer executes line 50 next. However, if the IF relationship is false, the computer just executes the line following the IF. In this case, if $N1 = 0$ is false, the computer executes line 30 after line 20. Then it executes line 40 and goes back to line 10.

The IF is called a **conditional branch** because the program execution is said to branch or to go to a line which depends on the relational condition after the IF. This is the same concept as for the ON GOTO. However, the IF test allows you to write the conditional test in a more natural way than the ON GOTO case.

You can even specify a different line number for the program to go to if the test fails. That is, the program does not have to automatically execute the line following the IF if the test fails. You can do this with the ELSE. For example, add these lines

```
20 IF N1=0 THEN 50 ELSE 60
60 PRINT "PRODUCT=";N1*N2
70 GOTO 30
```

This new version of line 20 has an ELSE in it. Now if N1 is not 0, the program goes to line 60 after line 20. Line 60 prints the product of N1 and N2. Next line 70 directs the computer to go to line 30 and to print the sum.

When you use an IF test, remember that some numbers cannot be represented exactly, so this can lead to errors. For example, as you saw in Chapter 2, the square root of 50 cannot be exactly stored in the computer. Enter and run the following program

```
10 PRINT 50-SQR(50)*SQR(50)
20 IF 50=SQR(50)*SQR(50) THEN 50
30 PRINT "NO"
40 END
50 PRINT "YES"
```

Line 10 prints the difference between 50 and the square of the square root of 50. When you run the program, line 10 prints 7.E-12. Line 20 checks if 50 equals the square of its square roots. As you saw from line 10, the numbers are not exactly the same, so the computer does not go to line 50. Instead, it executes line 30 and print "NO".

If you are doing calculations, it may be best to specify that the difference is less than some amount. For example, change line 20 to

```
20 IF ABS(50-SQR(50)*SQR(50))<1E-11 THEN 50
```

The absolute value function is used in case the difference is negative. Now when you run the program, the computer will print "YES".

If a relationship is true, the computer returns the value of -1 and goes to the line number following the THEN. If the relationship is false, a value of 0 is returned and the computer goes to the statement following the ELSE. If there is no ELSE, the computer just goes to the next statement after the IF. For example,

```
10 IF 0 THEN 30
20 IF -1 THEN 50
30 PRINT "NO"
40 END
50 PRINT "YES"
```

```
RUN
YES
```

```
** DONE **
```

Since a zero is given for the relationship in line 10, the computer does not go to line 30. Instead, it goes to line 20. Since there is a -1 for the relationship, the computer goes to line 50 and prints "YES".

Now change line 20 to

```
20 IF 1 THEN 50
```

and run. You'll still see a "YES" printed. The reason this occurs is that the computer interprets as true any non-zero relationship. So a 1, 5, 3.23, -

this program, it takes about 22 seconds to complete execution. Now replace line 30 with

```
30 A=A-I
```

and the program runs in about 13 seconds. The program runs much faster with $A=A-I$ because the computer does not have to convert the number with all the nines of the original line 30 to its binary form. You can see that to speed up programs, you should use variable names whenever possible so that the computer does not have to convert a number into its binary form.

Another way of speeding up the program is by replacing line 50 with

```
50 IF I THEN 30
```

Now the program takes only 11 seconds to run. The test $I > 0$ is true when I is unequal to zero, i.e., it gives a result of -1. However, when $I = 0$, the test gives a result of 0. So the test " $I = 0$ " is the same as "I".

Guess My Number

Now let's look at a simple game using the BASIC commands you've learned. We'll have the computer think of a random number between 1 and 10. You'll have three guesses to find the number. The computer will give you clues as to whether your guess is too high or too low.

Enter and run the following program for guessing numbers. Note that you will probably not get exactly the same results because of the RANDOMIZE statement of line 10.

```
10 RANDOMIZE
20 RNUM=INT(10*RND)+1
30 INPUT "GUESS MY NUMBER?":GUESS
40 IF RNUM<>GUESS THEN 70
50 PRINT "YOU GOT IT!"
60 END
70 IF RNUM>GUESS THEN 100
80 PRINT "HIGH"
90 GOTO 110
100 PRINT "LOW"
110 TRIES=TRIES+1
120 IF TRIES<3 THEN 30
130 PRINT "SORRY, TOO MANY TRIES. THE NUMBER WAS";RNUM
RUN
GUESS MY NUMBER?5
LOW
GUESS MY NUMBER?10
HIGH
GUESS MY NUMBER?7
HIGH
SORRY, TOO MANY TRIES. THE NUM
```

```
BER WAS 6
```

```
** DONE **
```

```
RUN
GUESS MY NUMBER?3
LOW
GUESS MY NUMBER?8
HIGH
GUESS MY NUMBER?5
YOU GOT IT!
```

```
** DONE **
```

Line 10 randomizes the pseudorandom numbers for RND in line 20. Otherwise, all the 'random' numbers will start off such that RNUM=6,4,6,4,3,6,3, etc. Line 20 generates a random number between 1 and 10. In developing a program with random numbers, it's always a good idea to check some values. So temporarily, you might want to add lines

```
25 PRINT RNUM
27 GOTO 20
```

and watch some of the numbers. To stop the program, use FCTN and CLEAR, then remove lines 25 and 27. Just type the line number and press ENTER for each line you want to delete.

Line 30 asks you to input a number which is assigned to the variable GUESS. Line 40 checks if the random number is unequal to your guess. If it is unequal the computer goes to line 70. However, if the random number does equal your guess, the computer goes to line 50 and prints "YOU GOT IT!", and then the program ends with line 60.

If you didn't guess the number, line 70 checks if your guess was less than the random number. If so, the computer goes to line 100 and prints "LOW." If your guess was greater than the random number, the computer goes to line 80 after line 70 and prints "HIGH." Then the computer goes to line 110.

Line 110 adds 1 to the number of tries you've made at guessing the number. Line 120 checks if you've made less than 3 tries. If so, the computer goes back to line 30 so that you can try another guess. If tries equal 3, the computer executes line 130 and the program ends.

There are several ways you might like to try to improve the program. After the game ends, the computer could ask if you would like another game, so that you don't have to keep entering RUN after every game. Another improvement would be to have the computer keep track of how many games you've won and lost during a session of play. You could even program the computer to also display the name and high score of players.

There's one other important thing to realize in designing games. If the player loses, don't be too harsh. If you add a line like

```
130 PRINT "YOU DUMMY-YOU BLEW IT"
```

you'll make the player feel pretty bad. Remember, the goal of a game is to have fun.

The Average Way

As another example of INPUT, IF and GOTO, the following program calculates the average of numbers. The average of a group of numbers is found by dividing the sum of the numbers by how many numbers there are. Enter and run the program for the examples shown. The special number -999999 stops input and commands the computer to print the average.

```

10 INPUT "NUMBER=?":NUMB
20 IF NUMB=-999999 THEN 60
30 SUM=SUM+NUMB
40 ITEMS=ITEMS+1
50 GOTO 10
60 PRINT "AVERAGE=";SUM/ITEMS
RUN
NUMBER=?1
NUMBER=?2
NUMBER=?3
NUMBER=?4
NUMBER=?5
NUMBER=?-999999
AVERAGE= 3

```

**** DONE ****

```

RUN
NUMBER=20.35
NUMBER=-8.2
NUMBER=118.9
NUMBER=-999999
AVERAGE= 43.68333333

```

**** DONE ****

Line 10 asks you to input a number and assigns it to the variable NUMB. Line 20 gives you an easy way to end input of the numbers. When you have finished giving all the numbers, enter the special value of -999999. When the IF test of line 20 finds this number, the computer goes to line 60 and prints the average. If the IF test does not find the -999999, then the computer executes line 30 which adds the latest number input to the sum of the previous numbers. The sum of all the numbers is stored in the variable SUM. Then the computer executes line 40 which increases the variable ITEMS by 1. This variable keeps track of how many numbers you have

input. Line 60 calculates and prints the average by dividing the sum of the numbers, SUM, by how many were input, ITEMS.

You can use any special number you want in place of -999999. Just pick one that is not likely to occur during calculations. For example, another good choice is 1E99. This is such a big number that there is very little chance of its occurring in data.

You may wish to expand this program by having the program ask if you want to run again after the average is printed. Just be sure to set SUM and ITEMS to zero before doing another average. Remember that RUN sets all variables to zero but you will have to provide for this in your program if you don't start it with a RUN each time.

Control Your Loop!

Your computer has a couple of commands which will let you easily control loops. These are the **FOR** and **NEXT**. To see how these work, first delete any existing program with the NEW command. Enter the following program to print the numbers from 1 to 9 and their squares.

```
10 FOR I=1 TO 9
20 PRINT I;I*I
30 NEXT I
```

When you run this, you'll see

```
RUN
1  1
2  4
3  9
4 16
5 25
6 36
7 49
8 64
9 81
```

```
** DONE **
```

The FOR statement of line 10 sets up a loop. The NEXT statement of line 30 completes the loop. The variable I is called the **control variable** or **index variable**. The control variable can have any legal name. For historical reasons, the variable names I, J, K, L, M, etc. are commonly used for the loop variables. If the name of the variable doesn't matter, then you may want to use I, J, K, etc. or any other name you want. The FOR statement says to initially set I to 1. Line 20 is then executed and the value of I and the square of I are printed. Then line 30 is executed. Line 30 increases the value of I by 1 and checks if the loop is done. If not, the computer executes line 20 again. So the computer prints a 2 for I and then a 4 since $2*2=4$. The NEXT statement of line 30 is executed again and I is set to 3.

This process of increasing I by 1 and printing its square keeps going until I=9. Line 20 prints I=9 and the square of 9. The NEXT statement sets I=10, determines that the loop is done, and the program ends.

The number to the left of the word TO is called the **initial value**. In this example, the initial value is 1. The number after the word TO is called the **limit**. If the control variable exceeds the limit, then the loop ends. When the computer executes the NEXT statement, the control variable is **incremented**. In our example, the increment is 1 and so the control variable is incremented by 1 each time the NEXT is executed. The computer checks if the control variable exceeds the limit. If so, the statement following the NEXT is executed. If not, the statement following the FOR is executed.

To see the final value of I, do a PRINT I command and you'll see a value of 10 printed for I. The loop ended because $9 + 1 = 10$ exceeded the limit of 9.

Although the increment was +1 in this program, you can use any increment with the **STEP** option. To see this change line 10 to

```
10 FOR I=9 TO 1 STEP -1
```

and run the program. Now you get

```
RUN
 9 81
 8 64
 7 49
 6 36
 5 25
 4 16
 3  9
 2  4
 1  1
```

The STEP option increments the control variable in any increment you choose. The default option of STEP is +1, which means that if you don't explicitly give a value, the computer assumes STEP=1. So the statements

```
10 FOR I=1 TO 9 STEP 1
```

and

```
10 FOR I=1 TO 9
```

mean the same. Using STEP, you can select positive or negative increments, and even decimal steps. For example, try a STEP of -.5 and you'll see the control variable decrease in steps of -.5.

You can use any numeric expression or variable for the initial value, limit, and step. For example, try

```
STEP -1/2
```

and you'll get the same results as STEP -.5. Likewise, try

```
10 FOR I=1/2+1/2 TO 3*3 STEP .1
```

and you'll see the computer print the squares in increments of .1. This gives the same result as

```
10 FOR I=1 TO 9 STEP .1
```

The values used for the initial value, the limit and the increment of a FOR-NEXT loop are called parameters. A **parameter** is a quantity which controls something, but which can be changed. For example, the area of a circle equals pi times the radius squared. Pi is a constant whose value cannot be changed. However, if you are given the relations

$$A=2$$

$$Y=A*X$$

then the "A" is a parameter, since its value can be changed and yet it does control the value of Y. The value of a parameter is fixed during a calculation, but can be changed afterwards.

As another example of FOR-NEXT loops, enter and run this new program.

```
10 FOR I=1 TO 9
20 N=1
30 PRINT N;"X";I;"=";N*I
40 NEXT I
```

You'll see the multiplication table for 1's printed.

RUN

1 x 1 = 1

1 x 2 = 2

1 x 3 = 3

1 x 4 = 4

1 x 5 = 5

1 x 6 = 6

1 x 7 = 7

1 x 8 = 8

1 x 9 = 9

** DONE **

Now change the value of N in line 20 to

```
20 N=2
```

and run. In this case, you'll see the multiplication table for 2's printed.

How to Make a Nest Without Birds

Suppose you want to print the multiplication table for 1 through 9. Well, you could change line 20 to

```
20 N=3
```

and run the program; change line 20 to

```
20 N=4
```

and run the program, etc. However, since the computer is supposed to make life easy for you, why not let the computer change N? Just add lines 20 and 35 so that your program looks like this, and run it.

```
10 FOR I=1 TO 9
20 FOR N=1 TO 9
30 PRINT N;"X";I;"=";N*I
35 NEXT N
40 NEXT I
```

This type of program has **nested loops** in it. The term nested loops means that one loop lies entirely within another. In this program, the loop for N lies within the loop for I. The N loop is the **inner loop** and the I loop is the **outer loop**. The computer will print an error message if a nested loop does not lie entirely within another. For example, delete line 35, add

```
50 NEXT N
```

and run the program. After the computer prints the products from 1×1 to 1×9 , you'll see the error message

```
* CAN'T DO THAT IN 50
```

You can have more than two nested loops so long as they do not overlap.

Jumping In and Jumping Out

Let's go back to the previous version of nested loops by deleting line 50 and adding

```
35 NEXT N
```

Now this program will print out the products from 1 to 9, just as before. However, suppose you want to stop printing when $N=2$. You could change the upper limit of N from 9 to 2. But, in many types of programs, you will want to include an IF test. For example, add the line

```
32 IF N=2 THEN 40
```

and run the program. Now you'll see

```
RUN
1 x 1 = 1
2 x 1 = 2
1 x 2 = 2
2 x 2 = 4
1 x 3 = 3
2 x 3 = 6
1 x 4 = 4
2 x 4 = 8
```

```

1 x 5 = 5
2 x 5 = 10
1 x 6 = 6
2 x 6 = 12
1 x 7 = 7
2 x 7 = 14
1 x 8 = 8
2 x 8 = 16
1 x 9 = 9
2 x 9 = 18

```

```
** DONE **
```

The IF test of line 32 jumps out of the inner loop into the NEXT statement of the outer loop. This illustrates a very popular use of the IF in looking for some condition inside a loop and then exiting the loop when that condition is fulfilled.

While it's O.K. to jump out of an inner loop to an outer loop, it's not all right to jump into an inner loop from an outer. Likewise you should not jump into a single loop. For example, add the line

```
5 GOTO 20
```

and run. You'll see

```
RUN
```

```
1 x 0 = 0
2 x 0 = 0
```

```
* CAN'T DO THAT IN 40
```

This program doesn't work right because we're jumping into the I loop. The value of I was not initialized to 1, and that's why the factor of 0 is used for I. The computer assumes by default that any numeric variable is initially 0. Then when the computer got to line 40, it encountered a NEXT I without first having executed a FOR I statement. This caused the error message.

Another point to note when you use loops is that a change in the control variable inside the loop does affect the loop. For example, do a NEW to delete this program and enter the following program:

```

10 L=1
20 U=5
30 FOR I=L TO U
40 PRINT I
50 NEXT I

```

When you run this program, the numbers 1,2,3,4, and 5 are printed. Now add these lines and run.

```

44 IF I<3 THEN 50
48 I=5

```

```

RUN
  1
  2
  3
** DONE **

```

As you can see, changing the control variable inside the loop does change when the loop terminates. When I was less than 3, line 44 directed execution to line 50, which just continued the loop. However, when I=3, line 48 was executed and I was set to 5. Line 50 then did the NEXT I to increment I and found that I exceeded the upper limit of 5 and so the I loop ended.

Now let's see what happens when we try to change the initial value or limit. Change line 48 to

```
48 U=2
```

and run the program. You'll just see the number 1,2,3,4, and 5. Next change 48 to

```
48 L=5
```

and run. You'll still see 1,2,3,4, and 5. If you add a STEP and change it, there will again be no effect on the loop termination.

The reason you can't change the loop termination is that the values for initial value, limit, and step increment are stored in certain memory locations when the FOR statement is executed. The values stored in these locations are copied from the numeric expressions or variables used for the initial value, limit, and increment. After the FOR statement is executed, you can't affect the values stored in that location which contains the loop parameters.

As a final thing to learn about the FOR statement, change line 30 to

```
30 FOR I=L TO U STEP 0
```

and run. You'll get the error message

```
* BAD VALUE IN 30
```

because the increment is 0. While you will probably never write a line with a STEP of 0 intentionally, you may write a line like

```
30 FOR I=L TO U STEP S
```

where S is some variable in your program. If S is 0 when the FOR is executed, you'll see the "BAD VALUE" error message.

Actually, the initial version of line 20 as

```
20 N=1
```

was very inefficient programming. It was a waste of the computer's time to keep redefining the value of N to 1 over and over again. If you need to initialize a variable, do it outside a loop. The initializing of N to 20 inside

the loop was done just so you could see how this line was replaced by a nested loop.

Building Tables

One useful application of FOR-NEXT loops is in making tables. For example, the following program prints a table of the accumulated principal built up for different interest rates. You enter the starting principal, the lower interest rate in percent, the upper interest rate in percent, the step in interest rate between lower and upper interest rates, and the number of periods. Following the program is the table showing the accumulated principal for 10% and 12% interest rates. There are many other variations of this **format** you may try. The term format means the way that data is organized or presented.

```

10 INPUT "PRINCIPAL,LOWER PERCENT,UPPER
PERCENT,STEP,NUMBER OF PERIODS ?":PR,LP,UP,ST,NP
20 PRINT :::"PERIODS";TAB(12);"INTEREST RATES"
30 FOR I=LP TO UP STEP ST
40 COUNT=COUNT+1
50 PRINT TAB(10*COUNT+2);I;"%";
60 NEXT I
70 PRINT::
80 LP=LP/100
90 UP=UP/100
100 ST=ST/100
110 FOR PER=0 TO NP
120 COUNT=0
130 PRINT PER;
140 FOR IN=LP TO UP STEP ST
150 COUNT=COUNT+1
160 PRINT TAB(10*COUNT);INT(100*PR*(1+IN)^(IN*PER)
+.5)/100;
170 NEXT IN
180 PRINT
190 NEXT PER
RUN
PRINCIPAL,LOWER PERCENT,UPPE
R PERCENT,STEP,NUMBER OF PER
IODS ?1000,10,12,2,10

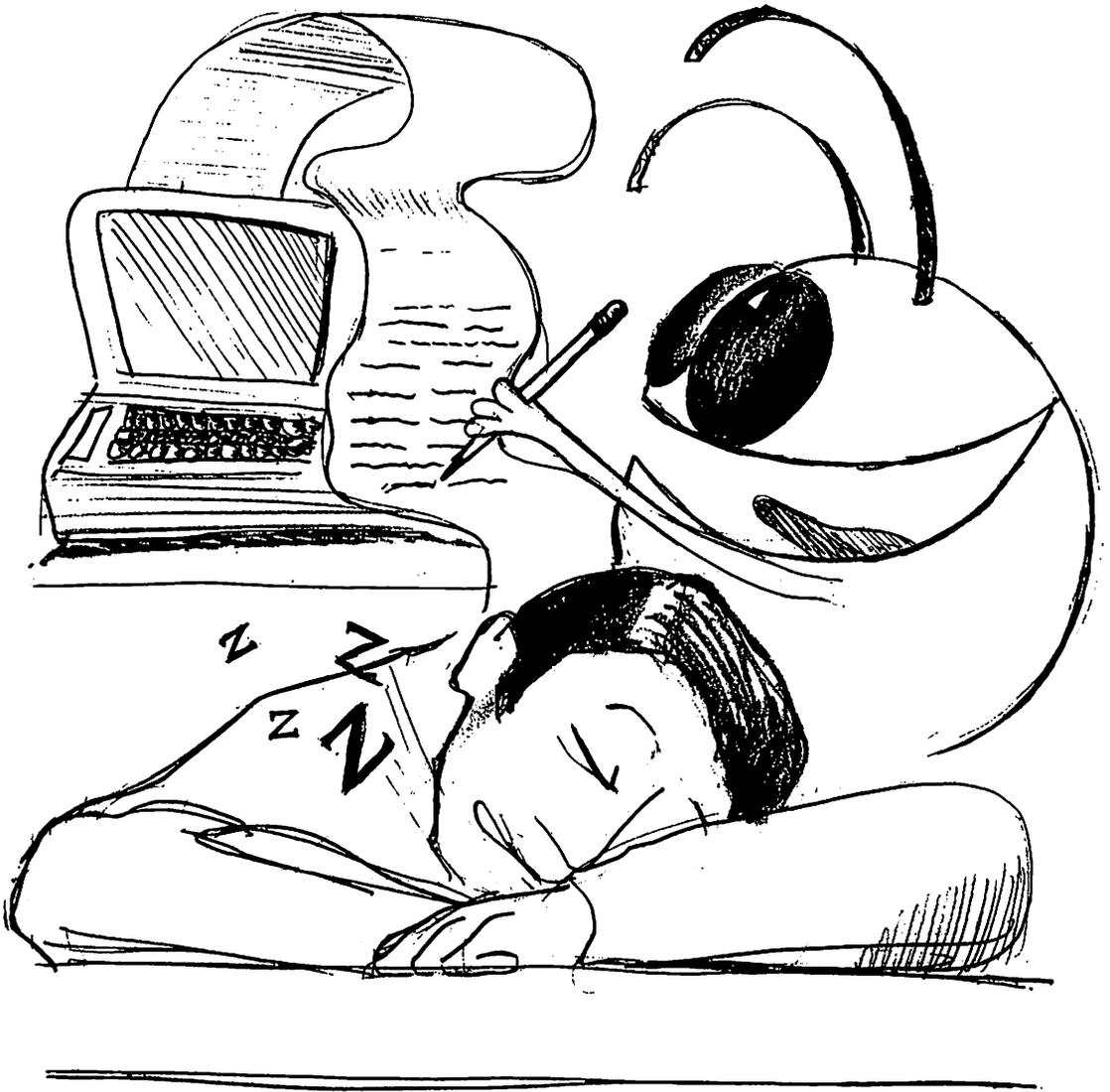
```

PERIODS	INTEREST	RATES
	10 %	12 %
0	1000	1000
1	1009.58	1013.69
2	1019.24	1027.57
3	1029.01	1041.64
4	1038.86	1055.9
5	1048.81	1070.36
6	1058.85	1085.02
7	1068.99	1099.87
8	1079.23	1114.93
9	1089.57	1130.2
10	1100	1145.68

Line 10 asks you to input the values needed by the program and assigns these values to variables. Line 20 prints the headings of the table. Lines 30-60 print the interest rates for each step. The variable COUNT is used for spacing the interest rates across the screen. Every time through the loop, COUNT is incremented by 1, so the next interest rate is tabbed over by $10 * 1 + 2 = 12$ spaces. Lines 80-100 convert the percent interests rates to decimal. The loop of lines 110-190 prints the table. The outer loop for PER sets the period for which the interest rates are calculated across. The inner loop of 140-170 calculates the interest rates for each period. Note that the accumulated principal is rounded off to two decimal places by the INT function.

If you run this program for more than two interest rates, the numbers will not all be in straight columns because there are only 28 columns on the TV screen. However, if you have a printer, you can modify this program to output to the printer and use the additional columns available on the printer.

You may also wish to modify this program to allow a user to enter the yearly interest rates and then have the program calculate the compound interest month by month, or year by year as an option.



“Poor fellow was so bored by the program he fell asleep. I’ll add some fun things to the program.”

Chapter 7

Getting To Know Your Dimensions

In this chapter, you'll learn some powerful ways to store and to manipulate data. You'll see how to apply these techniques in a practical application of maintaining a phone directory.

Names, Names, Names

Suppose you wanted to write a program to store and to manipulate hundreds or thousands of items. For example, you might want to keep track of

- items in a store—an inventory
- record or book collection
- mailing list
- telephone directory
- recipe collection
- accounts receivable

The collection of items is called a **database**. There are many computer programs designed to allow people to access and to maintain information in a database. An example that most people are familiar with is an airline reservation system. All the information about flights, schedules and passengers is contained in a database.

One question that you may be puzzled by is—how do the programmers manage to think of names for all those variables in the database? The answer is—they don't. Not only would it be hard to think up names—it would be even harder to write programs to manipulate the data. Instead, programmers use an efficient and powerful technique using **dimensioned variables**. A dimensioned variable is one which is accessed by a number as well as by its name. For example, enter the following program and run it as shown.

```
10 A(1)=1
20 A(2)=2
30 A(3)=3
40 PRINT A(1);A(2);A(3)
RUN
  1  2  3
** DONE **
```

The variables A(1), A(2), and A(3) are the dimensioned variables. A group of dimensional variables having the same name is called an **array**. You can use any legal variable name for the array. Each of the dimensioned variables is also called an **element** of the array. Each variable in the array is identified by an **index number** or **subscript**. This subscript is the number inside parentheses. Actually, the term inside parentheses can be a numeric expression or variable. The value of the subscript which identifies the variable is automatically rounded off to an integer by the computer. The terms array, element, and subscript come from math.

You can change the value of an array element just as for a **simple variable**. The term simple variable means one which is not dimensioned such as A, X, B1Z, AMOUNT and so forth. To change the value of A(1), change line 10 to

```
10 A(1)=-23.5
```

and run the program again. Now the new value of A(1) will be printed as shown below.

```
RUN
-23.5  2  3
** DONE **
```

However, in using dimensioned variables, you can't use the same name as a simple variable. For example, try to define a simple variable called "A" with the direct command

```
A=1
```

and you'll see the error message

```
* NAME CONFLICT
```

You can also use dimensioned variables with strings. For example, enter the direct command

```
N$(1)="TOM"
```

to define the dimensioned string variable N\$(1) as the string "TOM." Then do

```
PRINT N$(1)
```

and you'll see

```
TOM
```

printed since that is the value of N\$(1).

It's All in the Loop

The combination of FOR-NEXT loops and dimensioned variables is very powerful in manipulating data. Since subscripts are numbers, dimensioned variables can be easily accessed by a loop. For example, suppose

you want to assign the numbers 1 to 10 to the dimensioned variables. First, enter a NEW command and then enter and run the following as shown.

```

10 FOR I=1 TO 10
20 A(I)=I
30 PRINT A(I)
40 NEXT I
RUN
1
2
3
4
5
6
7
8
9
10

```

Lines 10 and 40 define the FOR-NEXT loop. Line 20 assigns the value of I to the dimensioned variable A(I). So A(1)=1, A(2)=2, A(3)=3, . . . A(10)=10. (Note that the three dots used in the preceding sentence mean to continue this assignment). Line 30 prints the value of each element A(I) after it is assigned a value. Actually, you can also store in the zero element. Change line 10 to

```
10 FOR I=0 TO 10
```

and run. You will see a 0 printed first since that is A(0)=0.

Of course, you can assign any values to the array. For example, change line 20 to

```
20 A(I)=10*RND+1
```

and run. You will see ten random numbers assigned to the array.

Let's Get Dimensioned

How many elements can you store in an array? Try increasing the size of your loop to

```
10 FOR I=0 TO 11
```

and run. The program will stop after the eleventh array element is printed with the error message

```
* BAD SUBSCRIPT IN 20
```

The reason this occurs is that your computer assigns a default limit of eleven elements for an array, such as A(0), A(1), A(2), A(3), A(4), A(5), A(6), A(7), A(8), A(9), A(10). If you want to use more than eleven, you'll need to

specify how many with a **dimension** statement. The dimension statement tells the computer how big your dimensioned array will be. This statement uses a BASIC word called **DIM** and must appear before any reference to the array. In our program, add

```
5 DIM A(50)
```

to allow 51 array elements. Note there are 51 because the zero element, A(0), also counts. Now you can print all 11 array elements when the program is run. In fact, you can increase the loop to

```
10 FOR I=0 TO 50
```

and print 51 elements.

There is an important thing to note about using dimensioned variables. When you use the DIM, the computer reserves space in memory for the dimensioned variables. The bigger the array size that you request, the more memory is required. What happens if you request more memory than the computer has? Let's try it and see. Change line 5 to

```
5 DIM A(2000)
```

and run. You'll immediately see the error message

```
* MEMORY FULL IN 5
```

because there isn't enough memory available. If you reduce the array size to 1000, the program will run.

The point of all this is that you should dimension only the number of elements needed by your program. In the example at the beginning of this section, only twelve elements were needed, yet 51 were dimensioned. That's a waste of memory. Although it may not matter for small sizes like this, you should get into the habit of being economical with memory. If you have a very large program, then it may not run if you do a DIM A(50) rather than exactly the amount you need.

You can use the DIM statement for string arrays also. However, there are no set amounts of memory specified for strings since the computer doesn't know in advance how long your string array elements will be. So each element of a string array is just set to the null string, i.e., contains no characters, until you assign characters.

Give Me a Name and I'll Tell You the Number

Now that you have an idea of how dimensioned variables work, let's give a practical application for a database program. The following program searches a database for a name you supply and then provides the phone number. Enter a NEW command, and then enter and run this program as shown.

```
10 N=3
20 DIM NAM$(3),PH(3)
30 NAM$(1)="JOHN DOE"
```

```
40 PH(1)=9990001
50 NAM$(2)="MARY SMITH"
60 PH(2)=9990002
70 NAM$(3)="JIM ADAMS"
80 PH(3)=9990003
90 INPUT "NAME? ":NAME$
100 FOR I=1 TO N
110 IF NAM$(I)=NAME$ THEN 150
120 NEXT I
130 PRINT "NAME NOT FOUND"
140 GOTO 90
150 PRINT PH(I)
160 GOTO 90
```

```
RUN
NAME? MARY SMITH
9990002
NAME? JOHN DOE
9990001
NAME? JOHN SMITH
NAME NOT FOUND
NAME? JIM ADAMS
9990003
NAME? JIMADAMS
NAME NOT FOUND
NAME?
```

As you can see, you must supply the exact spelling of the name in the database in order for the computer to find the number. Even the space between names is important, as the last example shows. To stop the program, press the FCTN and CLEAR keys.

Line 10 defines a variable, N, which stores how many names are in the database. Line 20 dimensions the names of people in the string array NAM\$(3), and the phone numbers in the numeric array, PH(3). Since only three names are stored, you could get by with no dimension statement. However you may want to expand this database beyond eleven names and so the DIM's are shown explicitly. Actually, you could also write this program using the 0 element. However, it's convenient to let N represent the total number of names in the database. If element zero were allowed, you'd have to use N + 1 as the total number of names.

Lines 30-80 contain the information in the database stored in the dimensioned variables NAM\$ and PH. The same subscript identifies the name and phone number of a person. For example, the subscript "1" identifies "JOHN DOE" and his phone number "9990001."

Line 90 asks you to input the name of the person whose phone number you want. This name is stored in the simple string variable NAME\$. Lines 100-120 make a FOR-NEXT loop to search the database for a match between the NAME\$ and each dimensioned string variable. If line 110

finds a match, the computer exits the loop and goes to 150. For example, if NAME\$="MARY SMITH", the computer sets I=1 and first tries NAM\$(1)="JOHN DOE". Since "JOHN DOE" does not equal "MARY SMITH", the computer continues through the loop with I=2. Now there is a match between NAM\$(2) and NAME\$, and so the computer goes to line 150. Line 150 prints the phone number PH(2) which corresponds to "MARY SMITH". Notice that when we exited the loop, the control variable I was equal to 2 since the names had matched. So line 150 prints the right phone number for "MARY SMITH". Then the program executes line 160 to go back to 90 and ask you for more input.

Suppose you ask for a name that's not in the database, like "JOHN SMITH"? The loop of lines 100-120 will terminate normally and then the computer executes line 130 to print "NAME NOT FOUND". Then line 140 directs the computer back to line 90 to ask for input again.

With this version, the only way to stop the program asking input is to do a FCTN CLEAR or QUIT, or turn the power off. However, you can easily provide a better way. Add the following lines

```
90 INPUT "NAME? (USE Q TO QUIT) ":NAME$
95 IF NAME$="Q" THEN 170
170 END
```

Line 95 checks if NAME\$='Q'. If so, the program goes to line 170 and ends. It's always a good idea to provide an orderly exit from a program, especially if the program may be used by non-programmers. They may not know that FCTN CLEAR will stop the program and will be annoyed because they can't stop the program. Also, notice that the information on how to quit is supplied by the INPUT. If possible, try to supply information like this in the input. It's much more convenient for a new user to immediately understand how to exit than to be given an instruction manual and to be forced to memorize all the commands your program accepts.

This consideration for the user will make your programs **user-friendly**. A user-friendly program is designed with consideration for the user. If you're writing just a short program for yourself, you may wish to skip user-friendly statements. However, it's really amazing how many programs that people write just for themselves are useful to others. It may take you less time to add some user-friendly instructions than to write up an instruction manual or to explain to every person what the commands are.

Add Some Strings

Because your computer prints up to ten digits of precision in numbers, you can even add the area code to the phone numbers. For example, change line 40 to

```
40 PH(1)=1234567890
```

and run the program for "JOHN DOE". You'll see the phone number "1234567890" printed out. The digits "123" would represent the area code.

However, international phone numbers are longer. Also, you may wish to include other information along with the phone number, such as the address. There's an easy way to extend our phone directory program so that it can handle any information. All you have to do is to replace the numeric variables with string variables. Since you can store any information in strings, you can include many more digits for the phone number, as well as the address.

Enter and run the following new program for the examples shown. You can obtain this program from the program in the preceding section by changing lines 20, 40, 60, 80 and 150 to match the following. The run is displayed as it appears on your screen.

```

10 N=3
20 DIM NAM$(3),INFO$(3)
30 NAM$(1)="JOHN DOE"
40 INFO$(1)="123-456-7890:1000 MAIN ST.,NEW
YORK,NY.,45637"
50 NAM$(2)="MARY SMITH"
60 INFO$(2)="15-213-777-5666:P.O. BOX
1789,TRENTON,NJ.,33314"
70 NAM$(3)="JIM ADAMS"
80 INFO$(3)="999-0003:APT.31,547 BROADWAY,LOS
ANGELES,CA.,67691"
90 INPUT "NAME? (USE Q TO QUIT) ":NAME$
95 IF NAME$="Q" THEN 170
100 FOR I=1 TO N
110 IF NAM$(I)=NAME$ THEN 150
120 NEXT I
130 PRINT "NAME NOT FOUND"
140 GOTO 90
150 PRINT INFO$(I)
160 GOTO 90
170 END
RUN
NAME? (USE Q TO QUIT) MARY S
MITH
15-213-777-5666:P.O. Box 178
9,TRENTON,NJ.,33314
NAME? (USE Q TO QUIT) JIM AD
AMS
999-0003:APT.31,547 BROADWAY
,LOS ANGELES,CA.,67691
NAME? (USE Q TO QUIT) Q

** DONE **

```

All the information about each person is stored in the string variable, INFO\$. The first set of digits in INFO\$ is their phone number. You can include as many digits as necessary, including hyphens (use the minus sign of your computer for a hyphen), to make it easier for you to read the number. After the phone number is a colon to separate this from the address. Of course, you could use a space or any other symbol instead of a colon.

Since INFO\$ contains string data, you can store anything you want. For example, NAM\$ could contain the names of food and INFO\$ could be the recipe. Or NAM\$ could be the names of recording artists and INFO\$ could contain all their records that you have. You can store and retrieve any type of items with a program like this.

Read My Data

If you have a lot of data to store, it takes a lot of typing for you to assign a value to every variable. Also, you use a lot of memory if you do only one assignment of a value to a variable in a line. Fortunately, your computer has some commands which make it easier for you to store and to access a lot of data. These are the **READ**, **DATA**, and **RESTORE** commands. To see how these work, type in the new program below and run it for the examples shown.

```

10 N=3
20 DIM NAM$(3),PH(3)
30 FOR I=1 TO N
40 READ NAM$(I),PH(I)
50 NEXT I
60 INPUT "NAME? (USE Q TO QUIT) ":NAME$
70 IF NAME$="Q" THEN 150
80 FOR I=1 TO N
90 IF NAM$(I)=NAME$ THEN 130
100 NEXT I
110 PRINT "NAME NOT FOUND"
120 GOTO 60
130 PRINT PH(I)
140 GOTO 60
150 END
160 DATA JOHN DOE,9990001,MARY SMITH,9990002,JIM
ADAMS,9990003

RUN
NAME? (USE Q TO QUIT) JOHN D
OE
  9990001
NAME? (USE Q TO QUIT) MARY S
MITH
  9990002

```

```

NAME? (USE Q TO QUIT) JIM AD
AMS
9990003
NAME? (USE Q TO QUIT) Q

** DONE **

```

In this version of the phone directory program, line 40 reads in data from line 160. The loop of lines 30-50 reads in all the data and assigns it to the dimensioned variables `NAM$` and `PH`. Notice that quotes are not necessary around the strings in line 160. The computer knows that if you're reading in a string variable, then the data is a string. Likewise if you're reading into a numeric variable, the data must be numeric.

You can have `DATA` statements anywhere in the program. When the computer executes the `READ`, it looks for the `DATA` statement with the lowest line number. The `READ` statement directs the computer to read in two values from the `DATA` and to assign those values to `NAM$(1)` and `PH(1)`. So "JOHN DOE" is assigned to `NAM$(1)` and "9990001" is assigned to `PH(1)`. The second time through the loop of lines 30-50, the values "MARY SMITH" and "9990002" are assigned to `NAM$(2)` and `PH(2)`. Likewise, the third time through the loop, "JIM ADAMS" and "9990003" are assigned to `NAM$(3)` and `PH(3)`.

How does the computer know which data to assign? The computer keeps track of the next value in the `DATA` statement that can be assigned using a **pointer**. The pointer is a memory location in the computer whose contents point to the address of the next item that can be read from the `DATA` statement. The term **address** refers to the memory location rather than the contents. For example, the string "JOHN DOE" may be stored in the computer's memory at address 1000. So the pointer contains "1000" rather than "JOHN DOE". Your TI-99/4A computer contains 16K addresses. Each byte of RAM is accessed by its address. BASIC automatically handles all the addressing for you. Initially, the pointer points to "JOHN DOE". After `NAM$(1)` is read, the pointer points to "9990001". After `PH(1)` is read, it points to "MARY SMITH" and so forth.

Where does the pointer point after all the data has been read? Let's try it and see. Enter the following lines to print the values assigned to `NAM$(I)` and `PH(I)`. The `STOP` will stop the program so you don't get asked for input.

```

45 PRINT I;NAM$(I);PH(I)
53 READ A$,B
55 PRINT A$;B
58 STOP

```

Line 45 prints the `NAM$(I)` and `PH(I)` inside the loop of lines 30-50. Then lines 53 and 55 try to read the next name and phone number.

When you run this program the computer will stop with the error message

```
* DATA ERROR IN 53
```

because there is no more data. After the values for the names and numbers were read in by the loop of lines 30-50, the pointer does not point to any more data and so the computer issues the error message.

However, you can reset the pointer back to the beginning of data using the RESTORE command. Enter this line

```
51 RESTORE
```

and run the program. You'll see

```
RUN
 1 JOHN DOE 9990001
 2 MARY SMITH 9990002
 3 JIM ADAMS 9990003
JOHN DOE 9990001
** DONE **
```

Because the pointer was reset by RESTORE to the beginning of the DATA statement, the values for "JOHN DOE" and "9990001" were read in for A\$ and B.

If you have more than one DATA statement in a program, you can do a RESTORE to any line by specifying the line number after RESTORE. To see how this works, let's first split up the single DATA statement of line 160 into three statements. Note that three statements increase the amount of memory used by the program and also mean more typing for you. In addition, the program takes longer to run since the computer must execute more lines. With all these considerations, you can see why it's better to make long DATA statements.

Make the following changes:

```
160 DATA JOHN DOE,999000
170 DATA MARY SMITH,9990002
180 DATA JIM ADAMS,9990003
```

Note that we could have made even six data statements by having separate ones for each value. You can also see how convenient it is to add DATA statements if they are placed at the end of the program. If you put them at the beginning or middle, you may have to change other line numbers of your program if you add a lot of data.

When you run the program, you'll see the "JOHN DOE" information printed because the RESTORE resets the pointer to the DATA statement with lowest line number, 160. This is the default form of RESTORE. Now let's change line 51 to restore to the second DATA statement by

```
51 RESTORE 170
```

Now when you run, you'll see the "MARY SMITH" information printed.

What happens if you give a line number for RESTORE that's not a DATA statement? Let's try it and see. Enter a

```
51 RESTORE 150
```

and run. You'll see the "JOHN DOE" information printed. So if you give a line number for RESTORE that's not a DATA statement, the computer will just go to the DATA statement with the next highest line number. Now suppose you give a line that's not even in the program, like

```
51 RESTORE 200
```

When you run this, you'll see the error message

```
* DATA ERROR IN 51
```

because there is no such line number.

Well, Tape My Data

The database program you've seen is fine if the data is not changed very often. You can just write the program with the data stored in it and run the program. However, if you want to change any information, you must change the program. Generally, it's not a good idea to change a program once it's working all right. Also, someone who's not a programmer would not be able to easily maintain the information.

A more practical database program will not store your data in the program itself. Rather, the program will read the data in from some other device such as a cassette recorder or disk. The data is kept in **datafiles** on the recorder or disk. A datafile is just a collection of information. It can be numbers or strings of characters. The information is transferred from the datafile into numeric or string variables in your program. Once the information is in your program, you can easily use it.

You can make any changes or searches of the database. When you want to stop using the program, the current data is written back to the cassette or disk.

Another advantage of using data not stored in the program is that you can have separate files for different kinds of data. For example, one datafile might contain names, telephone numbers and addresses of people. Another datafile might contain the names of records and recording artists in your record collection. Or the database could contain student names and grades; inventory items and amounts; meals and recipes, etc. The same database program can be used to maintain all these lists if the data is separated from the program. The alternative is a separate program for each type of data. However, this makes it hard for you to maintain the program. If you have ten separate programs and find a bug in one, you'll have to correct all of them.

The following database program is designed to work with a cassette recorder, although it can be easily modified for a disk. Consult your disk manual for information on storing and retrieving information. Enter and run for the examples shown. Note that when you input data containing commas you must enclose the input string in quotes. Also try the other commands to write and to read data from tape, etc. Note that in line 210, just press the quote key twice. Don't put any character between the quotes.

```

10 N=0
20 DIM NAM$(20),INFO$(20)
30 PRINT
40 PRINT "ADD DATA";TAB(17);1
50 PRINT "WRITE TO TAPE";TAB(17);2
60 PRINT "READ FROM TAPE";TAB(17);3
70 PRINT "SEARCH FOR NAME";TAB(17);4
80 PRINT "DELETE DATA";TAB(17);5
90 PRINT "PRINT ALL DATA";TAB(17);6
100 PRINT "END THE PROGRAM";TAB(17);7
110 INPUT CHOICE
120 IF CHOICE<1 THEN 30
130 IF CHOICE>7 THEN 30
140 ON CHOICE GOTO 190,290,360,460,550,700,150
150 INPUT "DO YOU WANT TO WRITE YOUR DATA? (Y OR
N)":RESPONSE$
160 IF RESPONSE$="N" THEN 180
170 GOTO 290
180 END
190 INPUT "NAME? (USE Q TO QUIT INPUT) ":NAME$
200 IF NAME$="Q" THEN 30
210 IF NAME$<>"" THEN 240
220 PRINT "YOU MUST GIVE SOME CHARACTERS FOR THE NAME"
230 GOTO 190
240 N=N+1
250 NAM$(N)=NAME$
260 INPUT "INFORMATION? ":INFO$(N)
270 PRINT N;"ITEMS STORED"
280 GOTO 190
290 OPEN #1:"CS1",SEQUENTIAL,INTERNAL,
OUTPUT,FIXED 128
300 FOR I=1 TO N
310 PRINT #1:NAM$(I),INFO$(I)
320 NEXT I
330 PRINT #1:"1E99","1E99"
340 CLOSE #1
350 GOTO 30
360 OPEN #1:"CS1",SEQUENTIAL,INTERNAL,
INPUT,FIXED 128
370 N=1
380 INPUT #1:NAM$(N),INFO$(N)
390 IF NAM$(N)="1E99" THEN 420
400 N=N+1
410 GOTO 380
420 N=N-1
430 PRINT N;"RECORDS READ"
440 CLOSE #1

```

```

450 GOTO 30
460 INPUT "NAME? (USE Q TO QUIT) ":NAMES$
470 IF NAMES$="Q" THEN 30
480 FOR I=1 TO N
490 IF NAM$(I)=NAMES$ THEN 530
500 NEXT I
510 PRINT "NAME NOT FOUND"
520 GOTO 460
530 PRINT INFO$(I)
540 GOTO 460
550 INPUT "NAME FOR DELETION? (USE Q TO QUIT) ":NAMES$
560 IF NAMES$="Q" THEN 30
570 FOR I=1 TO N
580 IF NAM$(I)=NAMES$ THEN 620
590 NEXT I
600 PRINT "NAME NOT FOUND"
610 GOTO 550
620 FOR J=I TO N-1
630 NAM$(J)=NAM$(J+1)
640 INFO$(J)=INFO$(J+1)
650 NEXT J
660 N=N-1
670 PRINT NAMES;" DELETED"
680 PRINT N;"RECORDS LEFT"
690 GOTO 550
700 FOR I=1 TO N
710 PRINT NAM$(I):INFO$(I)
720 FOR J=1 TO 300
730 NEXT J
740 NEXT I
750 GOTO 30
RUN

```

```

ADD DATA          1
WRITE TO TAPE     2
READ FROM TAPE    3
SEARCH FOR NAME   4
DELETE DATA      5
PRINT ALL DATA   6
END THE PROGRAM   7
? 1

```

```

NAME? (USE Q TO QUIT INPUT)
JOHN DOE
INFORMATION? 999-0001
  1 ITEMS STORED
NAME? (USE Q TO QUIT INPUT)
MARY SMITH
INFORMATION? "15-213-777-566

```

```

6:P.O. BOX 1789;TRENTON,NJ.,
33314"
  2 ITEMS STORED
NAME? (USE Q TO QUIT INPUT)
Q

```

When the program is run, you see a menu of choices displayed. This is a popular style of output because the user does not have to memorize all these commands. Simply select an item from the menu and input the number of your choice.

Line 10 initially defines the number of items, N, in the database as 0. As items are added or deleted, N will be adjusted correspondingly. Line 20 dimensions string variables for the names, NAM\$, and for the information strings, INFO\$. Although the example for this run are names, addresses and phone numbers, you can store any type of data in NAM\$ and INFO\$. Line 30 is just a PRINT to allow a blank line before the menu appears. This makes it easier for the user to read the display.

Lines 40–100 display the menu of choices for the user. Note that the numbers are displayed to the right of the text. This is easier for the user to remember than if the numbers were to the left of the text. Since people read left to right, then if the numbers were on the left, a user might read the number, then read the text and then have to refer back to the number again. Consideration of the user makes a program user-friendly.

Line 110 asks the user to input a choice while lines 120 and 130 check that the choice is a valid number from 1 to 7. This type of input checking is very important in a program which does not store data in program lines. Suppose the user accidentally inputs an "8". Without line 130, the program would stop with the error message

```
* BAD VALUE IN 140
```

You would not be able to continue and all the data you had input would be lost when you ran the program again.

Line 140 goes to the appropriate section of the program, depending on CHOICE. Lines 190–280 allow you to input data from the keyboard. Line 190 asks you for input of the name to be added. If you answer Q, the program assumes you're finished with input and takes you back to the menu. If you accidentally just press only the ENTER key, line 210 checks for this. If you input no characters, then the message of 220 is printed to tell you that some characters must be input for a valid name. Line 230 keeps taking you back to 190 to ask if you want to add more data. If you have entered some characters for the name, line 240 increases the value of the variable N, which tells how many names are in the database. Line 250 assigns the input name to the string variable, NAM\$(N). Then line 260 asks you to input the information about the name and stores it directly in INFO\$(N). Line 270 tells you how many items have been stored and line 280 takes you back to the input question.

Lines 290–340 write data onto cassette tape. Line 290 opens the cassette unit as file # 1 with an **OPEN** statement. For more detailed informa-

tion on cassette storage, consult your *TI-99/4A User's Reference Guide*. For disk storage, refer to the disk manuals. The cassette recorder that you normally use for loading in programs, CS1, is used for storage of data in this program. Actually, you can use two recorders. Use CS2 for writing output only, while CS1 can be used for both input and output. In this program, we'll assume you have only one recorder. Since a cassette recorder is used, the data are stored in a **sequential** manner. The term sequential means that each item of the data is stored one after another. The data are stored in an internal binary form, which is efficient for the computer. This is an output file and so the **OUTPUT** is specified. Also, since a cassette recorder is used, the data are of fixed length. That is, each time data are output, the computer always writes a certain number of positions on the tape.

The cassette recorder has a default of 64 positions for a record of data on tape. You can specify 128 or 192 positions. A number occupies 8 positions on the tape plus one more for a total of 9. This ninth position specifies the length, i.e., 8, of the number. A string occupies the length of the string plus one more for the length of the string. The loop of lines 300–320 print the two strings of name and information on to tape. So with FIXED 128, you have space for 126 characters. If you can get by with fewer characters, it is more efficient since you can use FIXED 64 or just FIXED by default. Only half as much tape will be used for storage and your datafile will take only half as long to read and write.

Line 330 prints a special string of "1E99" to mark the last data written to tape. When the computer reads the data back in from lines 380–410, it looks for the "1E99" in line 390. You can choose any string for this so long as it's not likely to occur in your data.

Line 340 closes the cassette file using **CLOSE**. It's important to always close open files or data might be lost from your program. Lines 360–440 input data from tape. Lines 460–540 do a search of the database for the name you input and print its information. Lines 550–690 delete data from the database. Note that the information is not deleted from the datafile on tape until you write the data back to tape. Lines 700–740 print all the data in the database to your screen. The loop of lines 720–730 acts as a time delay. It slows down the printing of data to the screen to make it easier for you to read. You may wish to change the value of 300 in line 720 to suit your reading speed.

If you have a printer hooked up to your computer, you can print out this information. So you could even use the program to print mailing labels. Enter these lines for printer output:

```
700 OPEN #1:"RS232.BA=1200"
705 FOR I=1 TO N
715 PRINT #1:NAM$(I):INFO$(I)
```

and the computer will also list to a printer via an RS232 interface at 1200 baud. If your printer operates at 300 baud, just change the "1200" in line 700 to "300", or whatever your printer speed is.

Many Dimensions

For the arrays we've been discussing so far, you can think of the array elements as lying in a straight line. To access the next element, add 1 to the subscript. To access the preceding element, subtract 1 from the subscript. In fact, an array with a single subscript to identify each element is called a **linear array**. Your TI BASIC actually allows up to three dimensions in an array. You can think of the elements of a **two-dimensional array** as cells in an area, while the elements of a **three-dimensional array** occupy a volume in space.

For a two-dimensional array, the first element specifies the row while the second specifies the column. Fig. 7-1 illustrates how you can think of these array elements from a geometric point of view for an array called N. Arrays with more than one dimension are called multi-dimensional arrays, where the prefix **multi** means many.

Enter and run the following new program to store numbers in a two-dimensional array:

```

10 DIM N(2,2)
20 FOR I=0 TO 2
30 FOR J=0 TO 2
40 N(I,J)=I+J
50 PRINT I;J;N(I,J)
60 NEXT J
70 NEXT I

```

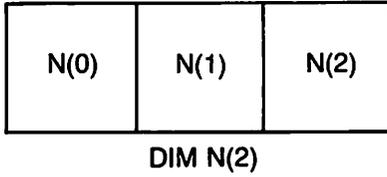
Line 10 dimensions the array. Although it's not necessary to dimension an array less than N(10,10), it's still a good habit for you to get into. Since the zero element is allowed, this is actually a 3×3 array because 9 elements are allowed. Lines 20–70 and 30–60 make up nested FOR-NEXT loops to provide the subscripts and data for the array. Line 50 prints out the row and column values for each array element, and also the value stored in the array.

In many real-life applications, you can use a two-dimensional array as a convenient way of storing data like:

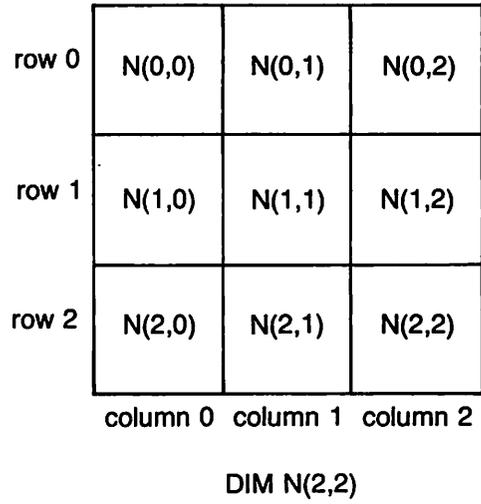
- salespeople's commission vs. type of item
- seating in an airplane, theater or other room
- inventory vs. department

The table below shows a two-dimensional array of student grades and subjects. The following simple database program stores the grades in a two-dimensional array and allows you to easily print out the grades and average of a student. Enter and run this program for the examples shown.

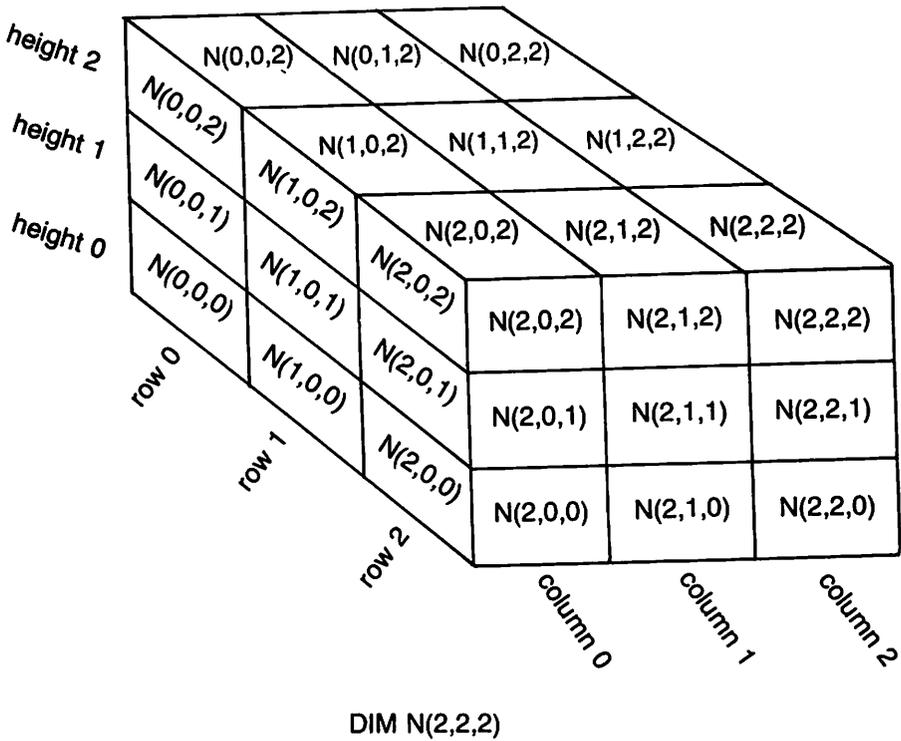
	COMPUTERS	MATH	ENGLISH	HISTORY	SCIENCE
John Doe	95	81	60	75	86
Mary Smith	93	95	85	90	92
Jim Adams	73	70	75	65	88



(a) One-dimensional array



(b) Two-dimensional array



(c) Three-dimensional array

Fig. 7-1 Geometric Concept of Arrays.

```

10 NUMSTUDENTS=3
20 NUMSUBJECTS=5
30 DIM GRADE(3,5)
40 FOR I=1 TO NUMSUBJECTS
50 READ SUBNAME$(I)
60 NEXT I
70 FOR STUDENT=1 TO NUMSTUDENTS
80 FOR SUBJECT=1 TO NUMSUBJECTS
90 READ GRADE(STUDENT,SUBJECT)
100 NEXT SUBJECT
110 NEXT STUDENT
120 PRINT
130 INPUT "STUDENT? (USE 0 TO QUIT)":STUDENT
140 IF STUDENT=0 THEN 220
150 SUM=0
160 FOR SUBJECT=1 TO NUMSUBJECTS
170 PRINT SUBNAME$(SUBJECT);TAB(11);GRADE
(STUDENT,SUBJECT)
180 SUM=SUM+GRADE(STUDENT,SUBJECT)
190 NEXT SUBJECT
200 PRINT "AVERAGE=";SUM/NUMSUBJECTS
210 GOTO 120
220 END
230 DATA COMPUTERS,MATH,ENGLISH,HISTORY,SCIENCE
240 DATA 95,81,60,75,86
250 DATA 93,95,85,90,92
260 DATA 73,70,75,65,88

RUN STUDENT? (USE 0 TO QUIT)1
COMPUTERS 95
MATH 81
ENGLISH 60
HISTORY 75
SCIENCE 86
AVERAGE= 79.4

STUDENT? (USE 0 TO QUIT)2
COMPUTERS 93
MATH 95
ENGLISH 85
HISTORY 90
SCIENCE 92
AVERAGE= 91

STUDENT? (USE 0 TO QUIT)0
** DONE **

```

Lines 10 and 20 define the number of students, NUMSTUDENTS, and number of subjects, NUMSUBJECTS. Line 30 dimensions the grades.

Lines 40–60 read in the names of the subjects and assign them to the string array SUBNAME\$. An alternative would be lines like:

```
42 SUBNAME$(1)="COMPUTERS"
44 SUBNAME$(2)="MATH"
```

and so forth. However, reading in the data like this takes up less memory. Lines 70–110 read the student grades and assign them to the two-dimensional array GRADE. Actually, GRADE could have been made a string array for the purpose of searching for a student grades. However, it's more convenient to make GRADE a numeric array if you want calculations, such as the averages.

Line 120 is just a PRINT to make the input requests easier for you to read. Line 130 asks for the student number. Note that this gives some degree of privacy to the student records. Someone who obtained this program without permission would not know who the students were since they are identified by number only. Of course, you could add an extra string variable to keep track of student names in the program.

Line 140 checks if you want to stop searching for student grades by seeing if you had input a zero. Lines 150–200 print the student grades and average. Line 210 takes you back to the input request.

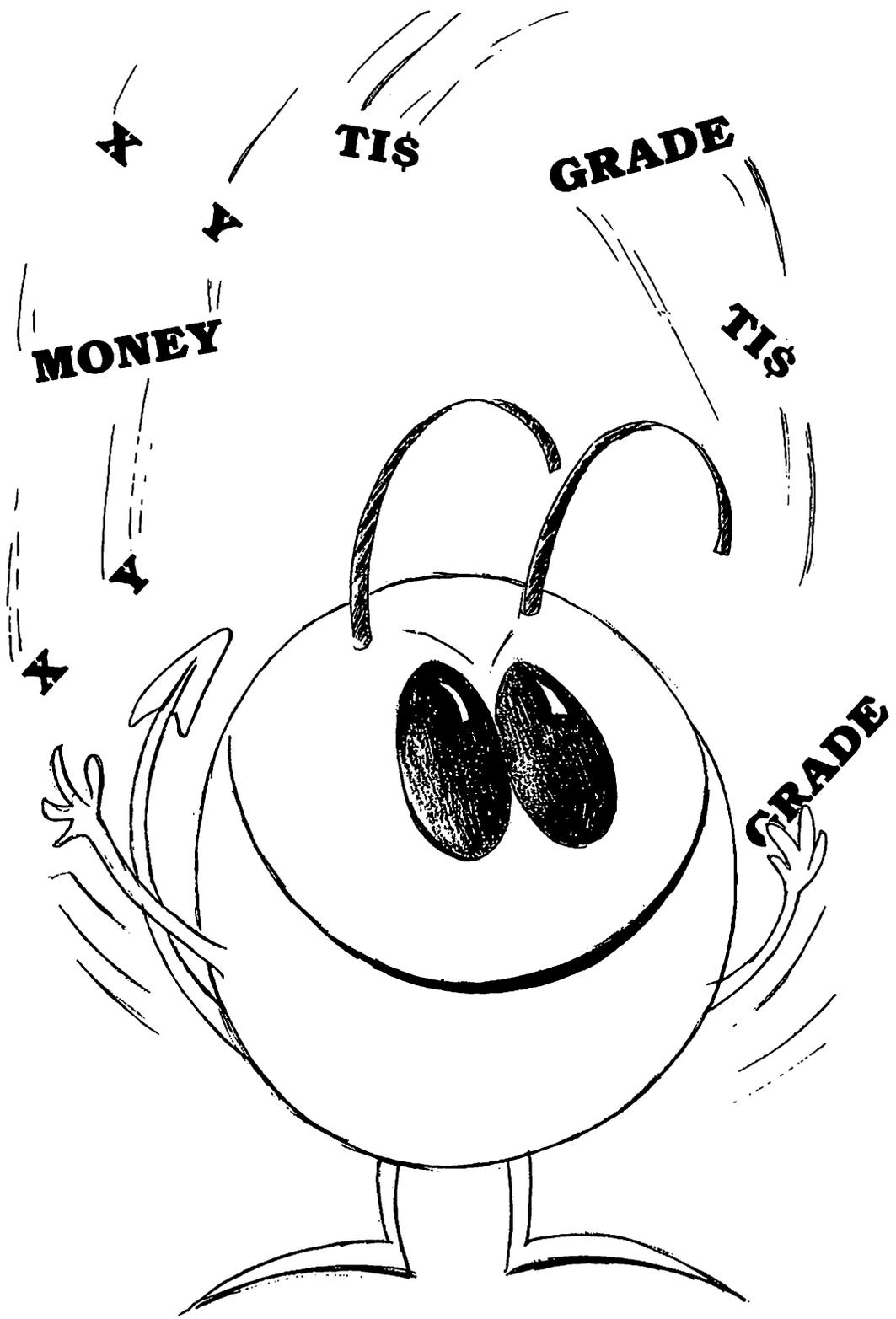
Of course, you could also use the database tape program to store the grades and compute the average if you make appropriate changes. For example, you could use numeric variables instead of string variables for INFO\$ in that program.

Changing Your Option

Another statement that is sometimes useful with arrays is the **OPTION BASE**. This allows you to set the lower limit of array subscripts at 1 instead of 0. If you aren't going to use the zero element of arrays, this option will save the memory space that is normally allocated to the zero element of an array. To use the OPTION BASE, include it as a statement with a lower line number than any DIM statement, or array reference. For example, in the previous two-dimensional database, use:

```
5 OPTION BASE 1
```

Note that the OPTION BASE applies to all arrays in the program. You can't have one array starting with a 0 subscript and another starting with a 1. Only one OPTION BASE is allowed in a program.



"I just love playing with variables."

Chapter 8

Debugging and Documentation

In this chapter, we'll cover some helpful statements and techniques to aid you in writing programs. These statements and techniques will save you time in designing and debugging your programs.

Ignore This Sign

When you write a program that you intend to save, it is helpful to include remarks in the program itself. For example, you could include the name of the program and a description of what it does. You might also include remarks at appropriate places in the program to tell what those sections of code do. The advantage of including remarks in the program is that they can't get lost. If you wrote the remarks on a piece of paper, you might lose it or have to look it up each time you want to change the program.

The following check balancing program includes the BASIC statement for remark at several places in the program. When the computer executes a line with the BASIC word **REM**, it will not try to execute the rest of the line. You can put anything after a REM and the computer will ignore it. The purpose of a REM is simply to provide documentation to whomever reads the program listing.

```
10 REM CHECK BALANCING
20 REM INPUT DATA
30 INPUT "INITIAL BALANCE? ":BALANCE
40 PRINT
50 INPUT "CHECK? (USE 0 TO QUIT) ":CHECK
60 IF CHECK=0 THEN 120
70 REM CALCULATE BALANCE
80 BALANCE=BALANCE-CHECK
90 REM PRINT NEW BALANCE
100 PRINT "BALANCE=";BALANCE
110 GOTO 40
120 END
RUN
INITIAL BALANCE? 1000

CHECK? (USE 0 TO QUIT) 100
BALANCE= 900

CHECK? (USE 0 TO QUIT) 89.9
```

```

BALANCE= 810.1
CHECK? (USE 0 TO QUIT) 10.1
BALANCE= 800
CHECK? (USE 0 TO QUIT) 13.98
BALANCE= 786.02
CHECK? (USE 0 TO QUIT) -100
BALANCE= 886.02
CHECK? (USE 0 TO QUIT) 22.92
BALANCE= 863.1
CHECK? (USE 0 TO QUIT) 10
BALANCE= 853.1
CHECK? (USE 0 TO QUIT) 0
** DONE **

```

When you list the program, you'll notice that the computer always puts two blanks after a REM. The computer does this to aid you in reading the program. If you type in only one blank after a REM, the computer adds in one. Likewise, the computer will automatically insert a space between INPUT or PRINT and the quote which follows in a statement.

Line 10 is the remark that gives the name of the program. Line 20 says that the next section of the program inputs data. Line 70 tells that the next part of the program calculates the balance. Line 90 remarks that the next program portion prints the output.

Shown following is a modification of the Check Balancing Program which prints all input and output to a printer operating at 1200 baud through an RS232 serial interface.

```

10 REM CHECK BALANCING
15 OPEN #1:"RS232.BA=1200"
20 REM INPUT DATA
30 INPUT "INITIAL BALANCE? ":BALANCE
35 PRINT #1:"INITIAL BALANCE?";BALANCE
40 PRINT
45 PRINT #1
50 INPUT "CHECK? (USE 0 TO QUIT) ":CHECK
55 PRINT #1:"CHECK? (USE 0 TO QUIT)";CHECK
60 IF CHECK=0 THEN 120
70 REM CALCULATE BALANCE
80 BALANCE=BALANCE-CHECK
90 REM PRINT NEW BALANCE
100 PRINT "BALANCE=";BALANCE
105 PRINT #1:"BALANCE=";BALANCE
110 GOTO 40
120 END

```

You may want to use remarks in a way similar to this: (1) title and description of program; (2) input; (3) calculations or other data processing; (4) output. However, keep in mind that remarks are not a substitute for good program documentation. In many programs, you need almost a line-by-line description of what's going on. It may be obvious to you what's happening, since you wrote the program. But a few weeks later, even you may be hard pressed to remember every detail of the program.

There is one final point to keep in mind regarding REM statements - they take up memory and slow down program execution. Even though the computer ignores the rest after the REM, it must first interpret the REM and find out what to do. It's as if you are driving down the freeway and keep seeing signs that say "Ignore This Sign." The message has no informational value to you, but it does distract you from driving, especially if you have to slow down to read the sign. In particular, it slows down the computer's execution speed if you have statements which do a GOTO to a REM, or include a REM in a FOR-NEXT loop.

A Good Design

A good computer program starts with a good design. Before you start to write a program, you should have a clear idea of

1. The required input and data needed by the program.
2. The calculations or other data processing required.
3. The form that you want to output.

Some people enjoy writing and debugging on a hit-or-miss basis. They like to experiment and to see if their solution to a bug works. Other people like to program because they want to produce a program that works correctly as soon as possible.

One way of designing programs uses a graphical aid called a **flowchart**. A flowchart is a diagram which describes the logical decisions and flow of data in a program. The flowchart uses symbols that have been defined by the American National Standards Institute (ANSI). These flowchart symbols are commonly available on plastic templates. You just run a pencil along the edge of the symbol on the template and draw it. Some computer programs are even available to draw flowcharts. You just input the program and the computer draws the flowchart. This is convenient since you can easily see the logical decisions and flow of data.

Fig. 8-1 shows a flowchart for the Check Balancing Program. Other symbols are available for disks, printer, etc. The lines and arrows show the direction of execution in the computer. Some people find flowcharts very helpful and others don't. However, flowcharts can be very helpful both in debugging and in designing a program. Before you write the BASIC code for a program, you can draw a flowchart to illustrate how the program should work. Likewise, if the program doesn't work correctly, a flowchart may aid in finding the bad logic.

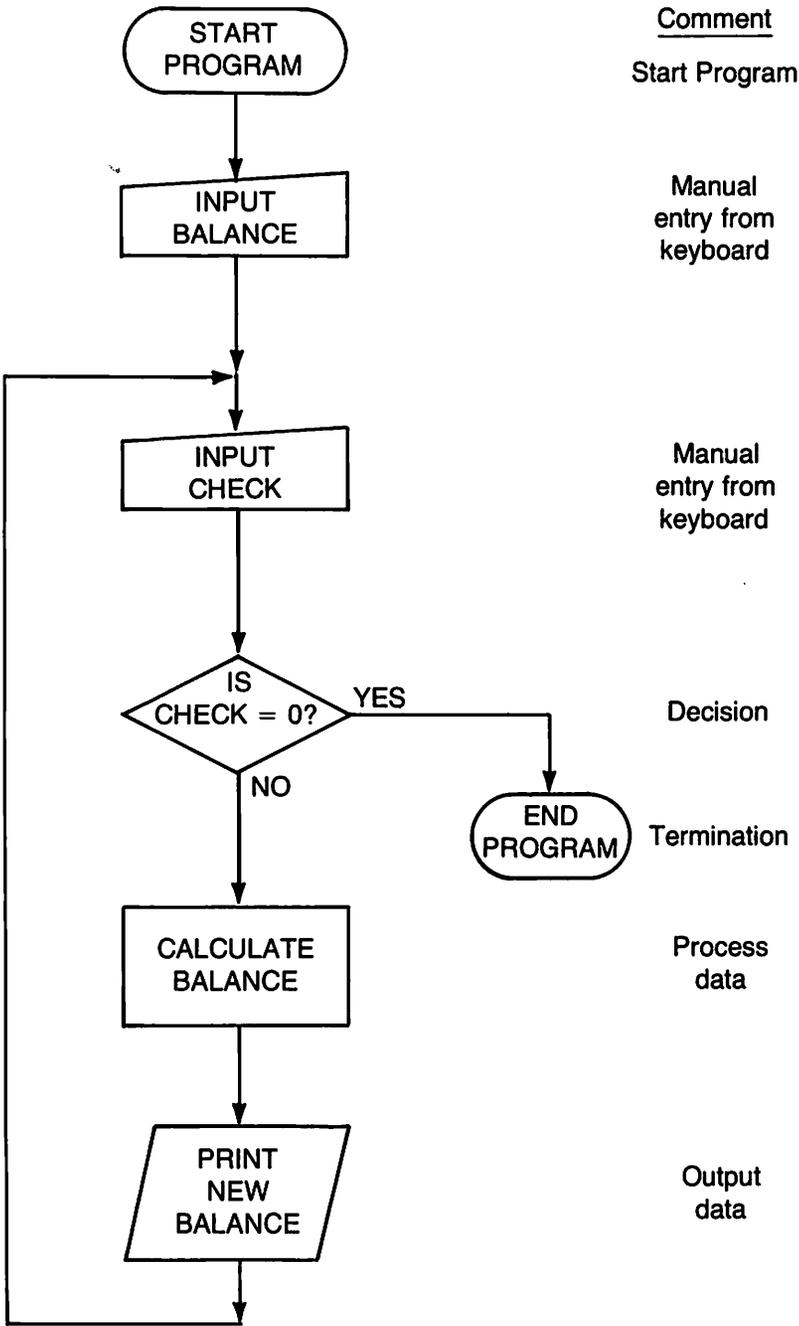


Fig. 8-1 Flowchart of the Check Balancing Program with Comments

Another approach to program design is **pseudocode**, which literally means **false code**. Pseudocode consists of English-like statements that describe the program. However, the description is usually not on as detailed a level as BASIC. The term **level** means the level of detail in the pseudocode description. A high-level is not very detailed, while a low-level is very detailed. Following are some pseudocode descriptions of the Check Balancing Program. You may wish to write the low-level pseudocode yourself from the program listing. The low level pseudocode will be almost a one-for-one translation of the BASIC statements into English.

Pseudocode Description of Check Balancing Program

Very High-Level

Write a program for check balancing

High-Level

Input the data

Process the checks

Output the results

Medium-Level

Input the starting balance

While there are checks being input

Begin

Calculate the new balance
by subtracting the check from the
balance and store the result as
the new balance.

End

In the medium-level pseudocode, the portion of the code between the Begin and the End is repeated as long as checks are being input, i.e., while CHECK is unequal to 0. For more information on program design and pseudocode, see the book *Foundations of Computer Technology*.

Follow That Line!

Your computer has some very useful features which will aid you in debugging. One of these is the trace command, **TRACE**. Type in TRACE and then press the ENTER key. Now run the Check Balancing Program (without printer output) for checks of 100, 89.9 and 0. You'll see the following

```
TRACE
RUN
<10><20><30>
INITIAL BALANCE? 1000
<40>
<50>CHECK? (USE 0 TO QUIT) 1
```

```

00
<60><70><80><90><100>
BALANCE= 900
<110><40>
<50>CHECK? (USE 0 TO QUIT) 8
9.9
<60><70><80><90><100>
BALANCE= 810.1
<110><40>
<50>CHECK? (USE 0 TO QUIT) 0

<60><120>
** DONE **

```

As you might have guessed, the TRACE feature traces the lines being executed and prints their numbers on the screen between the “<” and “>”. You can use TRACE either as a direct command or as a statement. It’s helpful to use TRACE as a statement when you want to trace the program after a certain line, rather than the entire program.

TRACE is particularly helpful in tracing down the lines in an infinite loop. For example change line 110 to

```
110 GOTO 100
```

and run the program for a balance of 1000 and check of 100. After the output is printed, the computer goes into an **infinite loop** between lines 100 and 110. The term infinite loop means the computer would keep looping forever. On the screen, you’ll keep seeing

```
<110><100>BALANCE= 900
```

You’ll have to interrupt the program with the FCTN and CLEAR keys to stop it.

Stop That Line

To turn off the TRACE feature if it was entered as a direct command, just use the untrace feature, **UNTRACE**. For example, type

```
UNTRACE
```

and press the ENTER key. You can also use UNTRACE in a program statement. By using TRACE and UNTRACE in certain portions of your program, you can selectively turn on and off the TRACE feature.

O.K., Break It Up

Another useful command for debugging is **BREAK**. Enter the following command

```
BREAK 40
```

and run the Check Balancing Program. Line 30 will ask you to input the balance. After you enter 1000 for the balance, the program will stop immediately with the message

```
* BREAKPOINT AT 40
```

As you can see, the BREAK followed by the line number 40 has halted program execution before line 40 is executed. If line 40 had been executed, the computer would have printed a blank line and then line 50 would have asked you to input the check.

Now that the program is halted you can print out the values of variables. For example do a PRINT BALANCE to see the value of BALANCE. You can execute any direct command that does not add, delete or change program lines. If you do not change any program lines, you can continue the program with a CONTINUE or CON, command. Enter a CON and you'll see the computer ask you to input a check amount.

You can have more than one breakpoint set by following the BREAK with a list of line numbers. For example

```
BREAK 40,60,80
```

is the same as entering the three commands

```
BREAK 40
BREAK 60
BREAK 80
```

Enter the BREAK 40,60,80 and just use CON to continue after every breakpoint until the program ends with a "*** DONE ***" message. Now run the program again without entering any breakpoints. This time you will see no breakpoints. The breakpoints set by a direct command BREAK are removed when you continue after the breakpoint. All of the breakpoints set by a direct command BREAK are also removed when a SAVE or a NEW command is entered.

You can also remove breakpoints with the **UNBREAK** command. Just enter UNBREAK to eliminate all breakpoints or follow UNBREAK by a list of the line numbers you wish to remove breakpoints. You can also include BREAK and UNBREAK as statements.

Chapter 9

Stringing Along

Your computer has many **string functions** which give you powerful programming power. In this chapter you'll learn the essentials of how to use these string functions.

What A Character

The **character function, CHR\$**, returns a character when you supply a numeric value for its argument. For example, enter the direct command:

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

and you'll see an "A" printed. The numbers that you can supply as arguments should be in the range of 0 to 127. Numbers in the range 32-127 correspond to the ASCII codes shown in the Appendix of ASCII Character Codes. The TI-99/4A computer has a special character assigned to code number 30. This character corresponds to the cursor block. Try

```
PRINT CHR$(30)
```

and you'll see a black square printed. The character with code 31 is another special character defined for your TI-99/4A computer. This is the edge character. The edge character looks like the cursor block. However, you cannot print it with the standard BASIC in your console. The TI Extended BASIC cartridge allows you to print the edge character and also provides many other features.

You can use a numeric expression or variable name in the argument of CHR\$. The value of the argument is rounded off to an integer when the CHR\$ function is used. If you look up the capital "A", you'll see it has a code of 65. To print a "B", just do a

```
PRINT CHR$(66)
```

Try the following program to show the first 61 characters. The asterisks are used to bracket each character so that you can see them better.

```
10 FOR I=0 TO 60
20 PRINT I;"*";CHR$(I);"*";
30 NEXT I
```

When you run this program, you'll notice that most character codes give blank characters. These are called **control codes** or **control characters**

because they are typically used to control devices such as printers, or used in data communications. Although the control characters are non-printing, the receiving device does respond to them.

Now change line 10 to

```
10 FOR I=61 TO 127
```

and run the program to see the rest of the characters.

What's The Opposite of a Character?

The opposite or **inverse function** of CHR\$ is the **ASC** function. Try

```
PRINT ASC("A")
```

and you'll see the number 65 printed. The ASC function returns the ASCII code number.

How's My Length?

The **length function**, **LEN**, returns the number of characters in a string. For example, enter this program:

```
10 INPUT "STRING? ":STRING$
20 PRINT LEN(STRING$)
30 GOTO 10
RUN
STRING? 1
1
STRING? A
1
STRING? ABC123
6
STRING? HELLO THIS IS ME
16
```

As you can see, the program prints out the number of characters in each string. Note that the numbers input are actually numerals. That is, the computer interprets a "1" or "123" as a string of numerals, not numbers. Numerals are symbols of numbers, but you can't use them in arithmetic. The computer internally stores a numeral differently from a number. You can't add the strings "1" and "123" any more than you can add the strings "APPLES" and "ORANGES".

For the example, "HELLO THIS IS ME", notice that spaces are counted as part of the string. In fact, the ASCII code for a space is 32. To enter just spaces, enclose them within quotes. For example, type " " and press ENTER. You'll see a 1 printed because you've entered one space. Next, press just the ENTER key when the computer asks for a string again. In this case, a 0 is returned since you entered no characters. As another example, hold down the control key, CTRL, press the "3" key once, and

then release the CTRL key and press the ENTER key. You'll notice the cursor does move over one space when you press CTRL and "3" since you've entered a control character. Although the control character is not printable, the computer recognizes its ASCII code and returns a value of 1 for the length.

You might like to try entering some more control characters and watching their output. For example, CTRL and the C key actually prints a squiggly line. CTRL and B also prints a symbol that looks somewhat like a man with a beard and mustache.

Now turn the power off and on your computer. Get into BASIC and press CTRL and C, and CTRL and B. You won't see any characters. During operation, the computer may assign some random pattern to a control character and this can be printed. However, the pattern can change and so you should not rely on a specific pattern.

You should be careful and avoid accidentally pressing the CTRL key instead of the SHIFT key. You might introduce a non-printing control character into your program. BASIC will tell you if the control character is part of a command. For example, enter

```
10 PRINT
```

and then press CTRL and the X key for the final character after the "T". This will introduce a non-printing control character after PRINT. When you press the ENTER key to enter this statement, the computer will respond with the error message

```
* BAD NAME
```

However, you can enter the string

```
10 PRINT "HELLO "
```

where the blank after the "O" in "HELLO" is actually a CTRL and X control character. If you are listing output to a printer, the control character may be interpreted as a command. Depending on the printer design, it may cause double characters, boldface printing, form feed, etc.

Give Me Your Position

Another useful string function returns the position of one string inside of another. For example, enter and run this program as shown. This program finds the position of the colon in the INFO\$ string, where position 1 is the leftmost. The **position function, POS**, is used to accomplish this.

```
10 INFO$="999-0001:1000 MAIN ST.,TRENTON,NJ.33344"
20 PRINT POS(INFO$,":",1)
RUN
9
```

The POS function has found the colon at position 9 in INFO\$.

The general form of POS has three arguments.

POS (data string, search string, starting position of search) where the data string is the one you want to search by the search string. The search will start at the starting position given by the third argument which can be a numeric expression or variable. If the search is not successful, then POS will return a value of 0. For example, change the search string colon in line 20 to a "\$" and run. A value of 0 is returned by POS because there is no "\$" in INFO\$.

Now change the starting position in line 20 to 50 and the "\$" back to ":". Run and you'll also get 0 because the starting position for the search exceeds the number of characters in the string. Finally, change the 50 to a -1 and run. Now you'll see the error message

```
* BAD VALUE IN 20
```

because a negative starting position was used.

Strings And Numbers

You can even convert numbers into strings with the **string function, STR\$**. Enter and run this program.

```
10 N=100
20 PRINT "THE NUMBER";N;"IS THIS"
30 PRINT "THE NUMERAL";STR$(N);"IS THIS"
RUN
THE NUMBER 100 IS THIS
THE NUMERAL100IS THIS
```

Notice that when STR\$ converts a number into a numeral, the leading and trailing blanks of the number are removed. That's why the numeral "100" runs into the strings before and behind it.

The string function is useful if you don't want the blanks around numbers. For example, enter the program to display the character codes. But this time, let's use STR\$ to pack more characters on a line by printing numerals.

```
10 FOR I=0 TO 60
20 PRINT STR$(I);"*";CHR$(I);"*";
30 NEXT I
```

In fact the characters are so much more packed now that you can see all but the top three lines if you change line 10 to

```
10 FOR I=0 TO 127
```

The Value of a String

Just as ASC and CHR\$ are inverse functions, so too are STR\$ and VAL. The **value function**, VAL, returns the number corresponding to a string of numerals. For example, try the following new program:

```
10 INPUT N$
20 PRINT VAL(N$)
30 GOTO 10
RUN
? 123
  123
? -123.4
-123.4
? -123.4E-5
-.001234
```

Notice that in the last example, the number is expressed as a decimal, rather than left in exponential form. As another example, stop the program with a FCTN and CLEAR, and enter the direct command

```
PRINT VAL("-123"&"E-3")
-.123
```

As you can see, you can perform a valid string operation so long as the argument reduces to a string of numerals or string expression of numerals.

Cutting Up a String

The final string function you'll see returns a portion or segment of a string. This is the **segment function**, SEG\$ function. For example, enter the following new program.

```
10 INFO$="999-0001:1000 MAIN ST.,TRENTON,NJ.33344"
20 P=POS(INFO$,":",1)-1
30 PRINT "PHONE:";SEG$(INFO$,1,P)
RUN
PHONE:999-0001

** DONE **
```

Line 10 defines a string variable with data for a phone number and address. A colon separates the phone number from the address. Line 20 finds the position of the colon in INFO\$ and subtracts 1 from it. Since the colon was at position 9, by subtracting 1 we now have the position of the last digit of the phone number string. This value of 8 is assigned to the variable P. Finally, line 30 extracts the string from position 1 to P from INFO\$ and prints it.

The general form of SEG\$ is
SEG\$(string, starting position, final position).

Shown following is a simple program using `SEG$` to reverse a word. The loop of lines 20–40 print the characters starting with the last position, `LEN (WORDS)`, to the first position. Actually, the program works with any string of characters, as you can see by “THIS IS A TEST.” example when the program is run.

```
10 INPUT "WORD? ":WORDS$
20 FOR I=LEN(WORDS$) TO 1 STEP-1
30 PRINT SEG$(WORDS$,I,1);
40 NEXT I
50 PRINT
60 GOTO 10
```

```
RUN
WORD? ABCDEF12345
54321FEDCBA
WORD? HELLO
OLLEH
WORD? THIS IS A TEST.
.TSET A SI SIHT
```

This same concept can be applied to our database program. Using the string functions, you can extract any portion of the string from `INFO$`. For example, if the database contained phone numbers and addresses, you could

- (1) extract all phone numbers.
- (2) extract numbers with certain area codes that you specify.
- (3) extract addresses in certain states or areas that you specify.
- (4) compile numerical statistics from the above data. For example, how many people live in a certain state or area code.

If the database contained students and grades, or inventory, you could extract any type of information. You may wish to enhance the Database Program some more using the above suggestions.

How To Get Organized

At some time in your life, you’ve probably had to organize a list of items by sorting them alphabetically. Perhaps it was a list of names, accounts, recipes, books, records. In any case, if you did it manually using index cards, you know what a boring and tedious job it was.

This is where your friendly, helpful computer comes into the picture. Computers just love long, boring and tedious work. Your computer can just zip through the sorting and leave you time for more creative work. As an example of sorting, let’s enhance the Database Program with some lines to sort by name in the database. When you add the lines to sort and `RUN`, the menu will show the additional entry

```
SORT ALL DATA      8
```

If you enter an 8 as your choice, the computer will sort the items in NAM\$ by alphabetical order and store the sorted items back in NAM\$. Of course, the corresponding items in INFO\$ will match the right NAM\$. You can use any of the other menu choices. For example, you can then print out the entire list by choosing a 6. You can also write the sorted items to tape using a 2. The sorted items are actually in order of their ASCII codes. So if your NAM\$ is "1234", it will be placed before "ADAM" because numerals precede letters. You can also include data with commas if you enclose the input in quotes. For example:

```
BAKER,TOM
```

will give an error message on input of NAM\$ because the comma separates items on a input list. The computer expects you to input only one item NAM\$. Instead, it appears you're inputting two items: (1) BAKER and (2) TOM. The way to get around this is to input the name in quotes

```
"BAKER,TOM"
```

Here are the lines to add sorting to the Database Program. Just type these lines in as shown after you've input the Database Program back into your computer from your cassette or other storage device.

```
100 PRINT "END THE PROGRAM";TAB(17);7:"SORT THE
DATA";TAB(17);8
130 IF CHOICE>8 THEN 30
140 ON CHOICE GOTO 190,290,360,460,550,700,150,1000
1000 PRINT "ITEM SORTED";
1010 NUMSORT=0
1020 FOR I=1 TO N
1030 ITEMNAM$=NAM$(I)
1040 ITEMINFO$=INFO$(I)
1050 FOR J=1 TO NUMSORT
1060 IF ITEMNAM$<NAM$(J) THEN 1090
1070 NEXT J
1080 GOTO 1130
1090 FOR K=NUMSORT TO J STEP -1
1100 NAM$(K+1)=NAM$(K)
1110 INFO$(K+1)=INFO$(K)
1120 NEXT K
1130 NAM$(J)=ITEMNAM$
1140 INFO$(J)=ITEMINFO$
1150 PRINT I;
1160 NUMSORT=NUMSORT+1
1170 NEXT I
1180 PRINT : "DONE WITH SORT"
1190 GOTO 30
```

Line 1010 defines a variable called NUMSORT which stores how many records have been sorted, where a record consists of a NAM\$ and an

INFO\$ for that item. The loop of lines 1020–1170 performs the sorting. Lines 1030–1040 define some variables to contain the data in the NAM\$ and INFO\$ to be sorted. The loop of lines 1050–1070 find out if the ITEMNAM\$ is less than the sorted item NAM\$(J). Notice that we're sorting items back into our original list. This takes much less memory than if we store all the sorted items in a separate list. If the name to be sorted is equal to or greater than an item in the sorted list, line 1060 goes to 1090. The loop of 1090–1120 inserts the item in the sorted list by copying all the sorted items up. Then 1130–1140 insert the items in the sorted list. Line 1150 prints the subscript of the item that was sorted. You may want to modify this to also print how many items there are in the list, which is stored in the variable N. If the item is greater than any item in the sorted list, the loop of lines 1050–1070 terminate and line 1080 goes to 1130. This just appends the item to the sorted list. Line 1160 increments the NUMSORT when another item has been sorted. Line 1180 prints a message telling when you're done with the sort and 1190 takes you back to the menu.

There are many ways to sort items. In fact, entire books have been devoted to sorting **algorithms**. An algorithm is a method for solving a problem in a finite number of steps. In fact, a computer program is an algorithm. Although the method used here is simple, more efficient methods are available which do not require so much moving of data. For example, see "Never Out of Sorts" by Doug Hapeman, pp. 16, in the *99'er Home Computer Magazine*, July 1983. Notice that every time we wanted to insert an item in the list, the loop of lines 1090–1120 had to move some data.

More efficient techniques use variables whose values are pointers. Just as the computer uses a pointer for READ, you can use the concept of a pointer for more efficient sorting. Rather than moving data around, you could rewrite this sorting to include numeric dimensioned variables as pointers that would point to an item. Since the TI BASIC in your console can't tell us the address of an item, the pointer contains the subscript of the array elements. Then just change the pointers to sort the data. For example, suppose your list of NAM\$ is

```
NAM$(1)="APPLE"
NAM$(2)="CANDY"
NAM$(3)="BANANA"
```

the original pointer list is

```
P(1)=1
P(2)=2
P(3)=3
```

After sorting, the pointer list would be

```
P(1)=1
P(2)=3
P(3)=2
```

To point out the sorted items, you would use the pointer list. For example:

```
FOR I=1 TO N
PRINT NAM$(P(I))
NEXT I
```

The original list of data is the same. Only the pointers have been changed to print the list in a different order. If you had an inventory with thousands of items, it would take too long to move the data around. Pointers would be necessary.

Scrambled Animals

Now that you've seen how the string functions operate, let's look at a fun, yet educational application. The following program uses string functions to scramble up the letters of a word. For example, CAT might become TCA or TAC or CTA, etc. The program has a list of words and randomly selects one word from the list, scrambles its letters, and asks you to guess the word. Following is a listing of the program and some samples of its output. Since RANDOMIZE is used, you will probably not get these exact scrambled words. Also, note that the following listing and examples are printed in this book with more than 28 characters on a line in order to make it easier for you to read.

```
10 REM SCRAMBLED WORDS
20 RANDOMIZE
30 PRINT "THESE ARE ANIMAL NAMES"
40 N=5
50 DIM WORD$(1000),W(50)
60 FOR I=1 TO N
70 READ WORD$(I)
80 NEXT I
90 TRIES=0
100 WORDNUM=INT(N*RND+1)
110 IF OLDWORDNUM<>WORDNUM THEN 130
120 GOTO 100
130 OLDWORDNUM=WORDNUM
140 LENWORD=LEN(WORD$(WORDNUM))
150 NUMLETTERS=0
160 FOR K=1 TO LENWORD
170 I=INT(LENWORD*RND)+1
180 IF NUMLETTERS=0 THEN 220
190 FOR J=1 TO NUMLETTERS
200 IF W(J)=I THEN 170
210 NEXT J
220 NUMLETTERS=NUMLETTERS+1
230 W(NUMLETTERS)=I
240 NEXT K
250 FOR K=1 TO NUMLETTERS
```

```

260 IF W(K)<>K THEN 290
270 NEXT K
280 GOTO 150
290 PRINT "SCRAMBLED WORD IS ";
300 FOR I=1 TO LENWORD
310 PRINT SEG$(WORDS$(WORDNUM),W(I),1);
320 NEXT I
330 PRINT
340 PRINT "GUESS MY WORD?(USE G TO GIVE UP;S TO STOP
PROGRAM)"
350 INPUT GUESS$
360 IF GUESS$="G" THEN 420
370 IF GUESS$=WORDS$(WORDNUM) THEN 440
380 IF GUESS$="S" THEN 460
390 PRINT "SORRY THAT'S NOT IT"
400 TRIES=TRIES+1
410 IF TRIES<3 THEN 300
420 PRINT "THE WORD IS ";WORDS$(WORDNUM)
430 GOTO 90
440 PRINT "YOU GOT IT!":
450 GOTO 90
460 END
470 DATA CAT,DOG,TIGER,LION,ZEBRA

```

RUN

THESE ARE ANIMAL NAMES

SCRAMBLED WORD IS TCA

GUESS MY WORD?(USE G TO GIVE UP;S TO STOP PROGRAM)

? TAC

SORRY THAT'S NOT IT

TCA

GUESS MY WORD?(USE G TO GIVE UP;S TO STOP PROGRAM)

? CAT

YOU GOT IT!

SCRAMBLED WORD IS OGD

GUESS MY WORD?(USE G TO GIVE UP;S TO STOP PROGRAM)

? GOD

SORRY THAT'S NOT IT

OGD

GUESS MY WORD?(USE G TO GIVE UP;S TO STOP PROGRAM)

? ODG

SORRY THAT'S NOT IT

OGD

GUESS MY WORD?(USE G TO GIVE UP;S TO STOP PROGRAM)

? GDO

SORRY THAT'S NOT IT

THE WORD IS DOG

```

SCRAMBLED WORD IS RGITE
GUESS MY WORD?(USE G TO GIVE UP;S TO STOP PROGRAM)
? TIGRE
SORRY THAT'S NOT IT
RGITE
GUESS MY WORD?(USE G TO GIVE UP;S TO STOP PROGRAM)
? TIGER
YOU GOT IT!

```

```

SCRAMBLED WORD IS OILN
GUESS MY WORD?(USE G TO GIVE UP;S TO STOP PROGRAM)
? LION
YOU GOT IT!

```

```

SCRAMBLED WORD IS ATC
GUESS MY WORD?(USE G TO GIVE UP;S TO STOP PROGRAM)
? G
THE WORD IS CAT

```

```

SCRAMBLED WORD IS IRGET
GUESS MY WORD?(USE G TO GIVE UP;S TO STOP PROGRAM)
? S

```

```

** DONE **

```

Line 20 randomizes the numbers from the RND function so that you won't always get the same series of pseudorandom numbers every time the program is run. Line 30 tells the user some information about the type of words used as data. In this program, line 470 contains all names of animals. This can be useful to the person playing since some permutations of the words give other valid words, where the term **permutation** means a rearrangement. For example, a permutation of "DOG" is "GOD". However, the computer is expecting "DOG" and will not accept the word "GOD".

Line 40 defines the variable N as the number of DATA words in line 470. If you add more data, just increase N. Line 50 dimensions the string array WORDS to allow up to 1000 words. Depending on the number of characters in the words, you may or may not be able to get that many in the program. The numeric dimensioned variable, W, is used to keep track of the permuted letters. It is set to 50 because it's unlikely you'll ever find a word with 50 characters in it. However, you can even scramble a phrase with this program. For example, include:

```

THIS IS A PHRASE

```

instead of CAT and you could get

```

AEASS PISI RTHH

```

So this program can even be used to encode messages. In that case, you may want more than 50 characters so just increase W accordingly.

Lines 60–80 read the data and assign them to WORD\$(I). Line 90 sets the number of tries you can make at guessing the word to zero. Line 100 generates a random number to pick one of the words in the list. Line 110 checks if the previous word number, OLDWORDNUM, is different from the new number, WORDNUM. This test is done so that the computer can't give you the same word twice in a row. If the old and new numbers are the same, line 120 directs the computer to go back and try again for a new WORDNUM. If the old and new word numbers are different, line 130 sets the old word number to the new word number.

Line 140 defines a variable for convenience, LENWORD, to contain the length of the word that the computer has randomly chosen. Line 150 initializes the variable NUMLETTERS to zero. This variable keeps track of how many letters in the random word have been permuted by the loop of lines 160–240. This loop generates the permuted numbers in the variable W. For example, the result of permuting DOG might be

```
W(1)=3
W(2)=1
W(3)=2
```

which indicates the third letter of DOG becomes the first, the first letter becomes the second, and the second letter becomes the third. The subscripts of W are the order in which the permuted word is written. The values assigned to W are the positions of the original letters. So the above represents "GOD".

Lines 250–280 prevent the same permutation as the original letters from occurring. For example, the random numbers might give

```
W(1)=1
W(2)=2
W(3)=3
```

and so "DOG" would be permuted as "DOG". This wouldn't be much of a challenge, so this loop directs the computer back to try again if the loop executes successfully. If the program gets to 280, then all the W(K)=K, so the word has not really been permuted.

Lines 300–320 print the permuted word by extracting a one-character substring from the WORD\$. The W(I) tells from which character position to extract the substring. Lines 330–380 ask you to input a guess, and also check if you want to give up or stop the program. Line 400 increments the tries by one each time you make a wrong guess. Line 410 checks if you have made less than three tries. If so, you get another chance. If you make three tries, the computer tells you the word, and then goes back to get another word. If you guess the word, line 440 prints a message of congratulations.

You can improve this program by adding variables to keep track of how many words you got right. You could display the percentage of correct guesses. You may also want to modify this program so that it will read in data from tape or disk files. Another variation you could try would be to modify this into a "Hangman" type program.

Chapter 10

Economize, Economize

In this chapter, you'll learn how to economize computer memory and your time in programming. Smaller programs also execute faster because the computer has fewer lines to execute. Your computer has some features that allow you to reduce the memory requirements of a program. At the same time, it takes less work by you to enter program lines.

Going Down Under

One way to save memory and your time is to write a program that re-uses portion of the code. This can easily be accomplished through use of subroutines. A **subroutine** is a group of statements which are accessed by a **GOSUB**, and which end with a **RETURN**. To see why subroutines are useful, enter the following simple program which does not use a subroutine. Then we'll write a new version with a subroutine and you'll see the difference.

```
10 INPUT N1,N2
20 ANSWER=N1+N2
30 GOTO 90
40 INPUT N1,N2
50 ANSWER=N1-N2
60 FLAG=1
70 GOTO 90
80 END
90 PRINT "ANSWER=";ANSWER
100 IF FLAG=0 THEN 40 ELSE 80

RUN
? 2,2
ANSWER= 4
? 1,5
ANSWER=-4

** DONE **
```

This program is designed to allow you to enter two numbers, find their sum, enter two more numbers, and find their difference. Notice that the program attempts to save memory by using line 90 to print the answer for both the addition and subtraction of the numbers.

However, the program must keep track of which line to go back to by means of a variable called FLAG. When FLAG=0, the computer goes back to line 40 so that you can enter the two numbers for subtraction. Then line 60 sets FLAG=1, prints the answer for subtraction, and the program goes to line 80 and ends. Although this approach works, it is inconvenient. It becomes even worse if you want to return to a third, a fourth or more places in the program.

A good solution to this problem uses a subroutine. Instead of your having to write the program in such a way that the computer goes back to the appropriate place, the computer does it. Enter and run the following new program using the subroutine statements GOSUB and RETURN. Notice how much simpler this approach makes the program. No flags are needed, and you can have as many GOSUB's as you want.

```

10 INPUT N1,N2
20 ANSWER=N1+N2
30 GOSUB 80
40 INPUT N1,N2
50 ANSWER=N1-N2
60 GOSUB 80
70 END
80 PRINT "ANSWER=";ANSWER
90 RETURN

```

When the computer executes a GOSUB, it first stores the memory address of the next statement it was going to execute and then starts executing from the line number following the GOSUB. This procedure is called a **subroutine call**. Another way of stating this is that the subroutine is being called by the **main program**. The main program is the portion of the program that is the main control of the program. In the example above, the main program consists of lines 10-70. In fact, you can write programs in which the main program is just a sequence of subroutine calls.

This address of the next statement to be executed is stored in a special area of the computer's memory called the **stack**. The term stack arises because these return addresses are arranged in the order that the subroutine calls are made. This situation is like the way dishes can be stacked on top of one another. To get to the bottom dish (the oldest subroutine call), the computer must first return from the most recent subroutine call. As new subroutine calls are made, i.e. more dishes are stacked, the return addresses are stacked on top of the most recent addresses. The RETURN statement must always be the last statement executed by a subroutine. The RETURN tells the computer to start executing from the top address of the stack. This top address and other information stored on the stack tells the computer the next line number that was to be executed before the call had been made. So the computer starts execution from that next line. A subroutine can call other subroutines or even itself. Also, just like the GOTO, there is an ON GOSUB statement.

Let's Get Drilled

The following program illustrates subroutines for an educational application of an arithmetic drill. This program is designed to give a student problems in addition or subtraction. If you answer correctly, the computer says "YOU ARE CORRECT!" If you are wrong, the computer tells you the correct answer.

```

10 REM ARITHMETIC DRILL
20 RANDOMIZE
30 INPUT "MAXIMUM NUMBERS USED=?":MAX
40 PRINT "ADDITION";TAB(12);1
50 PRINT "SUBTRACTION";TAB(12);2
60 PRINT "EXIT";TAB(12);3
70 INPUT CHOICE
80 ON CHOICE GOSUB 110,180,100
90 GOTO 40
100 END
110 GOSUB 250
120 PRINT TAB(10);N1
130 PRINT TAB(9);"+";N2
140 PRINT TAB(10);"-----"
150 CORRECT=N1+N2
160 GOSUB 280
170 RETURN
180 GOSUB 250
190 PRINT TAB(10);N1
200 PRINT TAB(9);"-";N2
210 PRINT TAB(10);"-----"
220 CORRECT=N1-N2
230 GOSUB 280
240 RETURN
250 N1=INT(MAX*RND+1)
260 N2=INT(MAX*RND+1)
270 RETURN
280 INPUT "ANSWER=? ":ANSWER
290 IF CORRECT=ANSWER THEN 320
300 PRINT "SORRY, THE RIGHT ANSWER IS";CORRECT
310 RETURN
320 PRINT "YOU ARE CORRECT!"
330 RETURN

```

```

RUN
MAXIMUM NUMBERS USED=?100

```

```

ADDITION      1
SUBTRACTION   2
EXIT          3
? 1

```

$$\begin{array}{r} 63 \\ + 65 \\ \hline \end{array}$$

ANSWER=? 128
YOU ARE CORRECT!

ADDITION 1
SUBTRACTION 2
EXIT 3
? 1

$$\begin{array}{r} 37 \\ + 21 \\ \hline \end{array}$$

ANSWER=? 68
SORRY, THE RIGHT ANSWER IS
58

ADDITION 1
SUBTRACTION 2
EXIT 3
? 2

$$\begin{array}{r} 57 \\ - 40 \\ \hline \end{array}$$

ANSWER=? -17
SORRY, THE RIGHT ANSWER IS
17

ADDITION 1
SUBTRACTION 2
EXIT 3
? 3

**** DONE ****

Line 30 asks you to input the maximum number used in the problems. For example, if you input a 100, the numbers can be integers from 1 to 100. Lines 40–60 print a menu of choices on the screen. You can select an addition or a subtraction problem, or exit from the program. Line 80 does an ON GOSUB to the appropriate portion of the program. If you select addition, the computer goes to line 110, subtraction to line 180, and exit to line 100.

Lines 110 and 180 each do a GOSUB to line 250. The subroutine starting at 250 generates two random integers and assigns them to N1 and N2. The computer then returns from the subroutine to the appropriate place in the subroutine that called it. For example, if the addition subroutine called it at 110, the computer returns to line 120. Lines 120–140 print the two random integers on the screen separated by a “+” sign. Then line 140 prints some minus signs to separate these numbers from the answer.

Line 150 calculates the correct answer for addition and assigns it to the variable CORRECT. Then the subroutine starting at 280 is called from

line 160. The subroutine at 280 is also used by the subroutine for subtraction. Both the addition and subtraction subroutines use 280–320 to ask for input and print output to you.

You may wish to try enhancing this program to add menu choices for multiplication and division. Also, you could assign ten problems at a time when a choice is made from the menu rather than returning to the menu every time after a problem. Another enhancement would be to keep track of the student's score for the ten problems.

Subroutines can be very useful in reducing program memory and in saving you time in typing in a program. You can even organize your program better by designing it using subroutines to accomplish specific tasks. However, subroutines do have a disadvantage in that they tend to slow down the computer. Since the computer must store the return address on the stack and later return, the computer must do more work. This slowing down will be more evident if a subroutine is inside a FOR-NEXT loop where it is called many times. In many types of programs, this slowing down will not even be noticeable to you. However, if you are designing certain types of games involving animation and moving objects, then execution speed may be more important to you than the advantage of subroutine.

How's Your Memory

In order to show that return addresses are stored, enter and run the following program. It takes about 10 seconds before execution stops. But first do a NEW to clear out any existing program.

```
1 A=A+8
2 GOSUB 1
RUN
* MEMORY FULL IN 1
```

This program is an infinite loop that keeps performing a GOSUB to itself. The subroutine calls itself at line 1 and stacks up its return address. Finally, the computer runs out of memory because so many addresses are on the stack!

Now do a

```
PRINT A
```

and you'll see the value 14536. Everytime a subroutine is called, eight bytes are needed to store the return address and other information on the stack. So A is actually the amount of memory that the computer used before running out of memory. The program itself uses 40 bytes and so the total amount of memory available for your programs is

```
PRINT A+40
14576
```

or 14,576 bytes.

Although the TI-99/4A does have 16K bytes = 16,384 bytes of memory, some of it is used by the computer for BASIC. The 14,576 represents the net amount of memory left for your program. So after you run this memory program, do a PRINT 14576-A to find out how much memory your program used. You can enter these lines 1 and 2 before any program lines to see how much memory your program requires for storage. For example, add this line

```
10 DIM B(1000)
```

and run. When the execution stops, do a PRINT A+40 and you'll see 6544 bytes of memory left. This means that every numeric dimensioned variable uses 8 bytes of storage since $14576 - 8 * 1000$ is about 6544 where the difference of 36 bytes comes from the length of line 10 and the tolerance of 8 bytes from our GOSUB memory check. Now change line 10 to

```
10 DIM B(1000),NS(1000)
```

and run. You'll see 4520. So each dimensioned null string takes $(6544 - 4520) / 1000$ or 2 bytes per string variable. Again the calculation above would be exactly 2 bytes if we included the storage for NS(1000) in line 10 and if the GOSUB memory check was more exact. However, for the most accurate estimate, include the two memory check lines after your program as the last lines to be executed. Of course, you'll need to change their line numbers to make them the last executed lines. It's best to put these lines as the last ones executed because the program uses memory beyond that required to store it. For example, dimensioned string arrays are null strings when a DIM is first executed. As the program executes, these strings will become filled with your data. So the size of your program will increase as it is run if you use dimensioned string arrays.

TI Extended BASIC has a command called SIZE that will show the amount of memory left and the amount used by the stack. However, you can always run this little program to check out the memory size in the standard TI BASIC that comes with your computer. Just don't name the variable used in this memory program the same name as a variable in your program or you'll get a Name Conflict Error Message. Also, remember that this program is accurate to within 8 bytes of the true amount of memory remaining.

Get Defined

Your computer has another way to save you typing and memory by allowing you to define your own functions. Just as SIN, COS, INT, are all predefined functions by the computer, you can have a **user-defined function**. That is, you can define your own function. Simply define your function with a **DEF** statement, as shown in the following program. Enter and run the program to print the square of a number, for the examples shown. Just use FCTN and CLEAR to stop.

```

10 DEF SQUARE(X)=X*X
20 INPUT "NUMBER=?":N
30 PRINT SQUARE(N)
40 GOTO 20
RUN
NUMBER=?2
4
NUMBER=?6.5
42.25
NUMBER=?90
8100

```

Line 10 defines your function with the DEF statement. Just follow DEF with the name of your function. If your function has an argument, it must be enclosed in parentheses as shown. Any variable name can be used as the argument in the DEF, so this is called a **dummy argument**. When SQUARE is used in line 30, just substitute the variable name or numeric expression for the dummy argument. For example, stop the program with the FCTN and QUIT keys, and then enter the direct command

```
PRINT SQUARE(2+2)
```

and you'll see 16 printed.

Of course, if you define your function in terms of a variable used in your program, then the function can be evaluated without an argument. For example enter and run the following version of the square program. You will see the same results as before if you use 2, 6.5, and 90 as inputs.

```

10 DEF SQUARE=X*X
20 INPUT "NUMBER=?":X
30 PRINT SQUARE
40 GOTO 20

```

You can define any function with a DEF, including string functions. Although the DEF is limited to a single statement function, it is convenient to use in many programs. You can also define string functions with a DEF.

In using a DEF, you need only include it with a line number lower than where you call its function. You don't actually have to execute the DEF line before you call its functions. Some limitations on the DEF are

(1) A DEF cannot be defined in terms of itself. For example, the following gives an error message if you try to run it.

```

10 DEF A(X)=A(X)+1
20 PRINT A(5)

```

Line 10 defines the function A in terms of itself. Then line 20 attempts to print a value. If you run this, you'll see the error message

```
* MEMORY FULL IN 20
```

(2) You can't define a DEF in terms of another. For example, the following gives an error message when run

```
10 DEF A(X)=B(X)+1
20 DEF B(Y)=2*A(Y)
30 PRINT A(4)
RUN
```

```
* NAME CONFLICT IN 20
```

However, subroutines can be defined in terms of one another. In fact, a subroutine can be defined in terms of itself.

Chapter 11

Give Me a Call

If you think your computer has some powerful built-in-functions like LOG, EXP, SIN, COS, etc.—wait—you ain't seen nothin' yet. In this chapter and the next, you'll see and hear just how powerful your computer is.

Now that we've covered the fundamentals of programming in BASIC, you can fully use the powerful graphics and sound features of your computer. You access these features by simply calling them with a CALL statement to the specific subprogram you want. The **subprogram** is like a subroutine in that it does its thing and then returns to the next statement following the one which called it. However, the subprograms are all predefined and you can't change them.

Clearing Things Up

The first subprogram we'll look at is so useful that it's usually entered as the first line of a program, after any introductory remarks. Enter and run the following:

```
10 CALL CLEAR
20 FOR I=1 TO 20
30 PRINT I
40 NEXT I
50 GOTO 10
```

When you run this program, line 10 will clear the screen. Lines 20–40 will print the numbers 1 to 20. Then line 50 will go back to line 10 and clear the screen again. This process can go on forever, since the program is an infinite loop. To stop, use the FCTN and CLEAR keys.

The **CALL CLEAR** subprogram fills the screen with blanks (ASCII code of 32). So CALL CLEAR is very useful as the first line of a program. By clearing the screen, your program's output starts fresh.

Getting Keyed Up

The **CALL KEY** subprogram is another feature available with your computer. It is very useful when you write game programs because the computer keeps executing the program while checking to see if you press a

key. In contrast, the INPUT statement halts the program execution at the line where INPUT is located.

The following program shows the numbers that are returned by the CALL KEY in a program. Enter and run this program.

```

10 CALL CLEAR
20 CALL KEY(0,K,S)
30 PRINT K;
40 IF K<0 THEN 50 ELSE 70
50 PRINT
60 GOTO 20
70 PRINT CHR$(K)
80 GOTO 20

```

When you first run the program you'll see a column of -1's marching up the screen. Now press the "A" key quickly and let go. You'll see a 65 followed by an "A" appear. If you didn't see an "A", you were too quick. Hold the "A" key down longer. In fact the longer you hold the key, the more "A"'s will appear. Press some other keys and you'll see a number followed by the ASCII code for that number. If you press no keys, you'll just see the -1's.

In this program, line 20 executes the CALL KEY subprogram. The first argument in parentheses, called the **key-unit**, defines how the keyboard will be scanned by the computer for a press. Refer to the Appendix of Keyboard Mapping to see diagrams as to how the key unit sets up the keyboard. The second argument is the **return-variable** which is assigned the value returned by the keyboard scan. Any variable name can be used for the return variable.

The possible values for the key-unit are:

- 0 The keyboard is set to the same **mode** as the previous CALL KEY statement that was executed. The term mode means manner or way. This key-unit value of 0 returns the same values as mode 5. Also, some additional control characters are available for modes 0 and 5 that may not be shown in the Appendix of the *TI User's Reference Guide*. For example, the CTRL and the following number keys return the control codes as shown

"1"	177
"2"	178
"3"	179
"4"	180
"5"	181
"6"	182
"7"	183
"0"	176

A key-unit of 0 gives compatibility with the older TI-99/4 computer. On that machine, only a key unit of 0 was available. So if you use a key-unit of 0 or 5 on your TI-99/4A, you can use pro-

grams for the TI-99/4. However, there are additional control characters available if you compare with the TI-99/4A. Compare Fig. 3 of the Appendix (Key-unit=5) with Fig. 1 (Key-unit=3). The key unit=3 on the TI-99/4A imitates the TI-99/4 keyboard. Notice how many more control characters are available on the TI-99/4A than the TI-99/4.

- 1 Only the left side of the keyboard or the left joystick is active.
- 2 Only the right side of the keyboard or the right joystick is active. Note that by alternatively using key-units of 1 and 2, you can give turns to two players. The following key-unit values all return upper case and lower case values, function codes, and control key codes. The values for each key are shown in the keyboard maps in the Appendix.
- 3 Standard TI-99/4 mode.
- 4 **Pascal** mode. Pascal is another computer language that is available as an option for your computer.
- 5 BASIC mode. This is the standard mode for the TI-99/4A.

Try running the program again with

```
20 CALL KEY(3,K,S)
```

and check the values returned for K with those in the Appendix for a key-unit of 3.

The third argument in the CALL KEY statement is the **status-variable**. To see how the status works, change line 30 to

```
30 PRINT K;S;
```

and run. As the following example shows, depending on your reaction time, you may get more or less output.

```

                Comment
-1  0
-1  0
-1  0
 57  1 9 (The "9" key is pressed)
 57 -1 9
 57 -1 9
 57 -1 9 (The "9" key is released)
-1  0
-1  0
-1  0
 40  1 ( (SHIFT is held down and then the "9" key
 40 -1 (   pressed)
```

Notice that the same physical key, "9", gives different return value when used with the SHIFT key. After the computer executes a CALL KEY, the computer returns one of the following three values for the status variable:

- +1 a different value was returned in the return variable since the last time a CALL KEY statement was executed. For example, if you pressed the X key, released it, and then pressed the C key, the status would change.
- -1 If you hold down the same key, the status changes to -1 after the initial status change to +1.
- 0 No key is being pressed.

As a simple example of CALL KEY, the following program makes blocks move up the screen. It looks like a worm race. Press the "1" key to tab the blocks to column 1, the "2" key to column 2 and so forth. Other keys will also move the blocks depending on their ASCII value.

```

10 CALL CLEAR
20 CALL KEY(0,K,S)
30 IF S=0 THEN 50
40 P=K-48
50 PRINT TAB(P);CHR$(30)
60 GOTO 20

```

The position variable P is initially zero. P represents the column number of the worm. Line 20 does a CALL key. Line 30 checks if a key is being pressed. If S=0 because no key is pressed, the program goes to 50 and prints the next block in the same column. Line 40 subtracts 48 from K so that if a "1" is pressed, a numeric value of 1 will be given to P. Likewise, if a "2" is pressed, P=2 and so forth for the other numeric keys.

Stop That Data!

Now that you've seen how CALL KEY works, let's use it in a practical application to control printing in the Database Program. When you pick choice 6 from the Database Program menu, the computer prints out all the data. Wouldn't it be nice if you could

1. stop scanning if you found something interesting?
2. scan forward toward the end of the data?
3. scan backward toward the beginning of data?
4. end printing all data and return to the menu?

Well, you can do all this by adding the following lines to the Database Program:

```

700 L=1
710 U=N
720 ST=1
730 PRINT "STOP SCAN-S":"SCAN BACK-E":"SCAN FORWARD-
X":"DONE PRINTING-D"
740 FOR I=1 TO 1000
750 NEXT I
760 FOR I=L TO U STEP ST

```

```

770 PRINT NAM$(I):INFO$(I)
780 CALL KEY(0,K,S)
790 IF S=1 THEN 840
800 FOR J=1 TO 300
810 NEXT J
820 NEXT I
830 GOTO 30
840 IF K=68 THEN 30
850 IF K=88 THEN 950
860 IF K=69 THEN 910
870 IF K<>83 THEN 990
880 CALL KEY(0,K,S)
890 IF S=1 THEN 840 ELSE 880
900 GOTO 990
910 U=1
920 ST=-1
930 L=I-1
940 GOTO 760
950 U=N
960 ST=1
970 L=I+1
980 GOTO 760
990 IF ST=-1 THEN 930 ELSE 970

```

Lines 700, 710 and 720 define initial values for the variables which control the lower limit—L, upper limit—U, and step size—S for scanning. Initially, the loop of lines 760–820 is set up to print items 1 to N. Item 1 is the first item you added while item N is the last item. Line 730 prints an explanation of the keys which control scanning. The loop of 740–750 introduces a time delay so that you can read the explanation. As you grow more experienced using these keys, you may want to reduce the “1000” in line 740.

Lines 780 and 790 check if a new key has been pressed. If so, the computer goes to 840. If no key has been pressed, the loop of 760–820 just keeps on printing items. Line 840 directs the computer back to the menu if the “D” key is pressed. If the “X” key is pressed, line 850 directs the computer to go to line 950 which sets up the loop parameters for line 760 to give a forward scan. If the “E” key is pressed, line 860 directs the computer to line 910 which starts setting up the loop parameters for line 760 to give a backward scan. If the “S” key is pressed, lines 880 and 890 keep looping until another key is pressed. Then the computer goes to 840 to find out if it was a “D”, “X”, or “S” key. If it was any other key, the scan just keeps continuing in the same direction it was going before it was stopped. Line 990 checks ST to see what direction the scan had been going.

Put It Where You Want

Until now, you’ve only seen how to print on the screen with PRINT. This is fine for many applications, but does make the output scroll up the

screen. In many games and programs, you may want to print a character at a certain place on the screen with no scrolling.

Your computer has two subprograms which allow you to easily print on the screen. These are the **CALL HCHAR** and **CALL VCHAR** subprograms. Enter and run the following program:

```

10 CALL CLEAR
20 FOR I=1 TO 24
30 FOR J=1 TO 32
40 CALL HCHAR(I,J,30,1)
50 NEXT J
60 NEXT I

```

As the program runs, you'll see cursor characters (black squares) fill up most of the screen with 24 rows and 32 columns. The code 30 for these squares is the third argument of HCHAR. After the program ends, the top three rows will have moved off the display after the **** DONE ****, a blank line, and cursor prompt are printed. You may have some trouble seeing some of the leftmost or rightmost columns if the picture is not quite centered on your TV set. The picture will look best on a video monitor. In designing programs with HCHAR or VCHAR, you may want to stick to the central portion of the TV screen for display and skip the leftmost two columns and rightmost two columns. Each of the squares represents a printable position of CALL HCHAR. The first argument of HCHAR is the row that the character will be printed at and the second argument is the column number. As you saw from the blocks starting at the upper left, this position corresponds to row=1, column=1. When the first row of blocks was completed, I=1 and J=32. When the last block was printed at the bottom right, it was at row=24, column=32. Fig. 11-1 shows a diagram of the block positions. This diagram is also called a **grid** because of the vertical and horizontal pattern of the lines.

Now change the program to the following by deleting lines 30 and 50 and editing line 40.

```

10 CALL CLEAR
20 FOR I=1 TO 24
40 CALL HCHAR(I,1,30,32)
60 NEXT I

```

When you run this program, you'll see the screen fill with blocks much faster than the previous version. While the previous version printed one block at a time, this program apparently prints a row at a time. Actually, the program still prints a block at a time, but does it so much faster that it looks like an entire row is printed simultaneously.

As you might have guessed, the secret to printing a row lies in the fourth argument of HCHAR. This fourth argument tells how many characters will be printed. That's why we could eliminate the J loop of lines 30 and 50. The J loop just printed a row of 32 blocks. Now the fourth argument takes care of that.

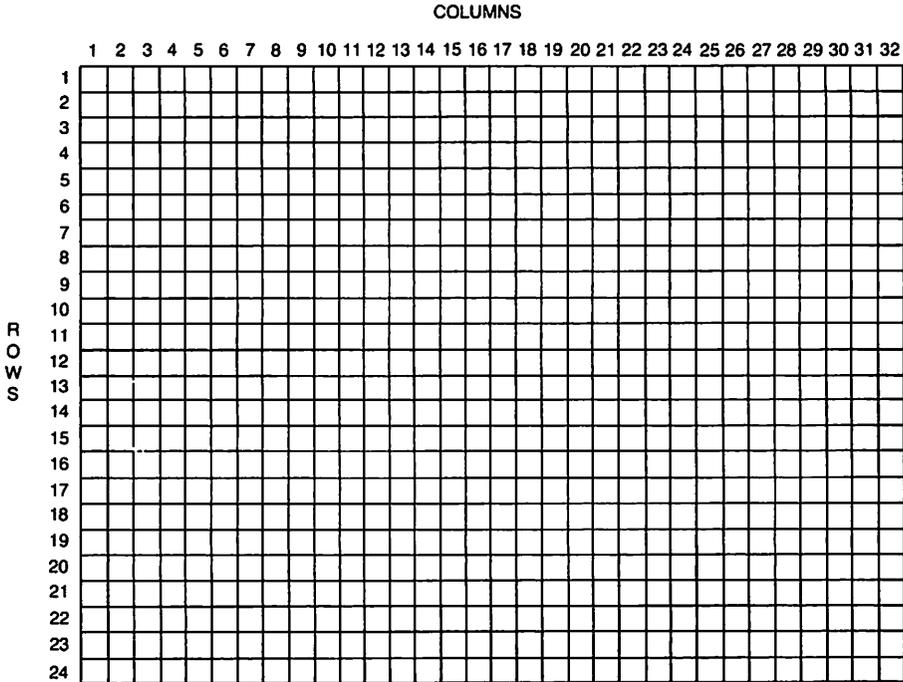


Fig. 11-1. Grid showing printable positions for CALL HCHAR and CALL VCHAR subprograms.

Change line 40 to

```
40 CALL HCHAR(I,1,30,10)
```

and run. Now only 10 blocks will be printed. Then, to make things easier to see, also move the blocks to the right by running with

```
40 CALL HCHAR(I,15,30,10)
```

Also, the fourth argument has a default option of 1. That is, the statements

```
40 CALL HCHAR(I,J,30,1)
```

and

```
40 CALL HCHAR(I,J,30)
```

mean the same to the computer. If the fourth argument for repetitions is missing, the computer assumes you mean 1 character. This same default option holds for CALL VCHAR.

Besides printing horizontally with CALL HCHAR, you can also print vertically with CALL VCHAR. Enter the direct command CALL CLEAR and then run the following program:

```
10 FOR J=1 TO 32
20 CALL VCHAR(1,J,30,24)
30 NEXT J
```

You'll see vertical columns printed from left to right. While the fourth argument of HCHAR repeats characters from left to right, the fourth argument of VCHAR repeats characters down the screen. If the bottom of the screen is reached and more characters are still to be repeated, they are printed starting at the top of the next column. Likewise, characters that go past the right screen boundary with HCHAR are printed on the next lower row starting from the left.

Now try the direct command

```
CALL HCHAR(1,1,42,768)
```

and you'll see the screen quickly fill with asterisks. Since HCHAR can keep printing on another row, you can supply more than 32 for the number of characters HCHAR will print. The same holds true for VCHAR. Try

```
CALL VCHAR(1,1,32,768)
```

and the screen will be erased as blanks, code 32, fill the screen. You can repeat anywhere from 0 to 32767 characters in HCHAR and VCHAR.

Also, the character codes repeat every 256 times. For example, try these 3 direct commands:

```
CALL HCHAR(24,1,65,10)
CALL HCHAR(24,1,321,10)
CALL HCAHR(24,1,577,10)
```

All print ten "A"'s at the lower left corner of the screen, just as

```
PRINT CHR$(65)
PRINT CHR$(321)
PRINT CHR$(577)
```

also print "A"'s.

You can actually use character codes up to 32767. However, they do repeat in groups of 256.

Draw Me a Picture

Let's use the CALL KEY and CALL HCHAR in a program that draws lines on the screen. The following program uses cursor blocks of code number 30 to make lines. The following keys are used:

```
"S" - left
"D" - right
"K" - up
"L" - down
```

Enter the following program. However, be careful in typing in the variable "COL". This variable consists of the letters "C", "O", and "L". Don't mistype a number zero "0" for the "O". Actually you can use a zero in "COL" and "ROW" if you're consistent, since variable names can include numbers. The problem arises if you use a zero in COL sometimes. Notice

that on your TV screen, the letter "O" has sharp edges and looks like a rectangle, while the number "0" has rounded corners. If you get an error message in this or any following programs that say BAD VALUE, its probably because you used "COL" or "ROW" in some places. Now run the program. Start off by pressing the keys briefly to keep the drawing in the central part of the screen. Then hold down the "K" key until the program stops with the error message

* BAD VALUE IN 180

because ROW=0.

```

10 REM DRAW #1
20 CALL CLEAR
30 ROW=12
40 COL=16
50 CALL KEY(0,K,S)
60 IF S=0 THEN 50
70 IF K=83 THEN 110
80 IF K=68 THEN 130
90 IF K=75 THEN 150
100 IF K=76 THEN 170
110 COL=COL-1
120 GOTO 180
130 COL=COL+1
140 GOTO 180
150 ROW=ROW-1
160 GOTO 180
170 ROW=ROW+1
180 CALL HCHAR(ROW,COL,30)
190 GOTO 50

```

Lines 30 and 40 define the initial values of the row, ROW, and column, COL, where you'll draw. These were picked to start off in the center of the screen. Lines 50-100 check for the key that you press. If no key is being pressed, then the status variable S=0 and the computer just goes back to line 50. The ASCII codes for the keys "S", "D", "K", and "L" are shown in lines 70-100.

If you press the "S" key, the computer goes to line 110, subtracts 1 from COL and then goes to 180. Line 180 prints the cursor block at the new position and then line 190 directs the computer back to the CALL KEY line. With this simple program, there is no error checking and that's why the program halted with an error message when you kept moving the cursor block up the screen. The HCHAR and VCHAR subprograms can print only to rows 1-24 and columns 1-32.

Let's introduce some error checking by adding the following lines. If you try to move past the screen boundaries with these added lines, the cursor will stop just at the boundary.

```

110 IF COL=1 THEN 50
115 COL=COL-1
130 IF COL=32 THEN 50
135 COL=COL+1
150 IF ROW=1 THEN 50
155 ROW=ROW-1
170 IF ROW=24 THEN 50
175 ROW=ROW+1

```

When you add these lines and run the program, you won't get an error message because the cursor can't go past the screen boundaries.

In fact, you can restrict the cursor to any area of the screen by changing the boundary parameters in lines 110, 130, 150, and 170. These boundary parameters are the numbers 1, 24, and 32 that the ROW and COL are compared to. For example, in line

```
110 IF COL=1 THEN 50
```

the "1" is the boundary parameter for the left side of the screen. When you design a program like this, it's usually convenient to leave the boundary parameters with variable names. By giving names, it's easy for you to change the parameters wherever they may be used in the program by changing the single line where the parameter is defined.

There is still another bug left in this program. Although it's designed to use only the "S", "D", "K", and "L" keys, you can move left by pressing any other key. What we really want is to ignore any key except "S", "D", "K", and "L". To accomplish this, add the line

```
105 GOTO 50
```

Now if a key is pressed that is not "S", "D", "K", or "L", line 105 will direct the computer back to 150.

One enhancement you can add to this program is to make the current position of the cursor visible. It's a lot easier to draw if you can see where you are drawing from. So add these lines to make the cursor blink at its current position.

```

50 CALL HCHAR(ROW,COL,32)
51 CALL HCHAR(ROW,COL,30)
58 CALL KEY(0,K,S)

```

The statement of line 50 will print a blank at the current cursor position. Line 51 will then print a block at the current position. The computer then executes the CALL KEY of line 58 and may update the row or column before going back to line 50. Because the computer does this so rapidly, the current cursor will appear to blink or flash very rapidly.

Another enhancement you can add is a cursor that erases. This is called a **destructive cursor** because it will erase characters that it passes over. Shown following is the new version of the program. To obtain this from your program, just replace lines 10 and 140, and add lines 56, 57, 102, 104, 106, 107, 108, 109, 120, and 160 from the following. Delete

line 180 and your program should look like the following, called the DRAW #2 Program.

```

10 REM DRAW #2
20 CALL CLEAR
30 ROW=12
40 COL=16
50 CALL HCHAR(ROW,COL,32)
51 CALL HCHAR(ROW,COL,30)
56 IF ERASE=0 THEN 58
57 CALL HCHAR(ROW,COL,32)
58 CALL KEY(0,K,S)
60 IF S=0 THEN 50
70 IF K=83 THEN 110
80 IF K=68 THEN 130
90 IF K=75 THEN 150
100 IF K=76 THEN 170
102 IF K=87 THEN 108
104 IF K=69 THEN 106
105 GOTO 50
106 ERASE=1
107 GOTO 50
108 ERASE=0
109 GOTO 50
110 IF COL=1 THEN 50
115 COL=COL-1
120 GOTO 50
130 IF COL=32 THEN 50
135 COL=COL+1
140 GOTO 50
150 IF ROW=1 THEN 50
155 ROW=ROW-1
160 GOTO 50
170 IF ROW=24 THEN 50
175 ROW=ROW+1
190 GOTO 50

```

When you start to run this program, press the "D" key and you'll see the cursor going to the right. Now press the erase key, "E". Hold it down for a second or two and you'll see the cursor blink slower. Now, press the "S" key to retrace the original cursor path. As you hold down the "S" key, the cursor erases the blocks. To stop this destructive cursor, press the write key, "W". Hold it down for a second or two until the blinking rate increases. Now when you move the cursor, it writes on the screen. You can switch back and forth between writing and erasing anytime by pressing the "W" and "E" keys.

The variable ERASE is used in this Draw #2 Program to tell the computer whether you want to write or erase. When ERASE = 1, the computer will erase. When ERASE = 0, the computer will write.

After the computer prints a blank in line 50 and a block from line 51, it checks the value of ERASE in line 56. If ERASE=0, the computer goes to line 58 and the block remains on the screen. However, if ERASE=1, it goes to line 57 and prints a blank before continuing to line 58. This gives the destructive cursor since the blank wipes out any character on the screen.

Line 102 checks if you've pressed the "W" key to write. If so, the computer goes to line 108 and sets ERASE=0 before going back to line 50. Line 104 checks if you've pressed the "E" key. If so, the computer goes to line 106 and sets ERASE=1. Also, notice that we've eliminated line 180 by using the blinking cursor lines of statements 50 and 51; and the new lines of 56 and 57. Although more statements are used now, the program can do more than before.

Play Me a Tone

One of the nicest features of your computer is its sound effects. You can easily include sound with the **CALL SOUND** subprogram. To add sound to the DRAW #2 Program, just add the line

```
52 CALL SOUND(100,110*ROW,COL/2)
```

Line 52 makes a tone. A tone is a sound of a single frequency. When you run the program, you'll hear a tone as the cursor blinks on the screen. Press the "D" key and you'll hear the tone become quieter in volume as the cursor moves to the right. Press the "S" key and it will get louder in volume as the cursor moves to the left. Press the "K" and the tone becomes lower in **frequency** or **pitch**. Press the "L" key and the tone becomes higher in frequency. The frequency of sound waves refers to the number of vibrations a sound makes per second. A higher frequency or pitch means more vibrations per second. One vibration per second is called one cycle a second.

The first argument of CALL SOUND tells the computer how long the tone should last. This is called the **duration** and is given in **milliseconds** where 1 millisecond = .001 second. The abbreviation for millisecond is **msec.** and so 1000 msec. = 1 second. Your computer can produce a sound duration from 1 to 4250 msec. In other words, it can produce tones from .001 to 4.25 seconds, where the actual duration may vary by up to 1/60 sec. From line 52, you can see that we have specified a duration of 100 msec. for the tone produced by CALL SOUND.

Change the tone duration to 500 in line 52 as

```
52 CALL SOUND(500,110*ROW,COL/2)
```

and run the program. Press the "K" for a few seconds to move the cursor up and then the "L" key to move it down. You'll hear the tone change in frequency as you change rows. Then change the duration to 1000. Notice that the blinking lasts longer now. Although the computer keeps executing

statements after a CALL SOUND is executed, it will stop at the next CALL SOUND if the previous tone is still playing.

You can get around this by using a negative duration. For example, use -1000 instead of 1000. This **negative duration** tells the computer to cut short the tone if it comes to another CALL SOUND statement and to start the new tone immediately.

The second argument of CALL SOUND specifies the frequency. You can select any frequency from 110 **hertz** to 44733 hertz. A hertz is abbreviated **Hz** and is equal to 1 cycle per second. The range of frequencies for someone with excellent hearing is about 20 Hz to 20,000 Hz. The actual frequency produced by the computer may vary from 0 to 10% of what you specify, depending on the frequency.

In line 52, the frequency varies with the row number. Since the top row is 1, the frequency of CALL SOUND is $110 * 1 = 110$ Hz. In the second row, the frequency is $110 * 2 = 220$ Hz, and so forth. You also have the option of giving a negative value from -1 to -8 for this argument and the computer will produce various types of noise. These noise effects are useful in some games. To hear these effects, enter in the following program without erasing the drawing program. Then do a RUN 1000 to hear the sound effects.

```
1000 FOR I=-1 TO -8 STEP -1
1010 PRINT I
1020 CALL SOUND(1000,I,2)
1030 NEXT I
1040 GOTO 1000
```

The program starting at line 1000 will show the noise argument and you'll hear the sound corresponding to it. To stop the program, use FCTN and CLEAR. Then delete lines 1000 to 1040.

The third argument of CALL SOUND specifies the tone **volume**, where 0 is loudest and 30 is quietest. The volume of a sound refers to how loud it is. If the argument exceeds 30 or is less than 0, you'll get an error message. For example, do a direct command

```
CALL SOUND(500,1000,1)
```

and you'll hear a tone. But

```
CALL SOUND(500,1000,-1)
```

or

```
CALL SOUND(500,1000,31)
```

will give the error message

```
* BAD VALUE
```

Don't confuse the tone sounded when the error message appears with the tone of CALL SOUND. The last two examples do not produce a tone, except for the error message tone.

You can actually specify up to three tones and one noise in a single CALL SOUND. For example,

```
CALL SOUND(1000,200,2,400,2,600,2,-3,2)
```

will generate three tones and one noise simultaneously. A sequence of arguments is duration, frequency of first tone, volume of first tone, frequency of second tone, volume of second tone, frequency of third tone, volume of third tone, frequency of noise, volume of noise where the duration applies to all the tones and noise. It is not necessary that the noise be in the fourth argument position.

Update Your Display

It's interesting to see the frequencies produced by CALL SOUND in the drawing program. Let's add some lines to display the frequency at a fixed location on the screen. Note that this can't be done with PRINT because the screen would scroll upwards. Add these lines

```
157 GOSUB 200
180 GOSUB 200
200 ROWSOUND$=STR$(110*ROW)
210 FOR I=1 TO LEN(ROWSOUND$)
220 CALL HCHAR(1,25+I,ASC(SEG$(ROWSOUND$,I,1)))
230 NEXT I
240 RETURN
```

and change line 52 back to

```
52 CALL SOUND(100,110*ROW,COL/2)
```

Lines 157 and 180 go to the subroutine to print the frequency whenever the row is changed. Since line 52 makes the frequency depend on the row, only those portions of the program which change the row need update the display of the frequency.

Line 200 converts the frequency to a string of numerals. For example, if ROW=10, then the frequency is $110 * 10 = 1110$. The string produced by line 200 is then "1110". It's necessary to produce a string of whatever you want printed because HCHAR and VCHAR will print only a single character of the string at a time.

Lines 210–230 print the string on the screen. The FOR-NEXT loop and SEG\$ extract each numeral one at a time. For example, from "1110", SEG\$ starts with the leftmost "1" and keeps extracting until the rightmost "0". The ASC function converts the "1" to its ASCII code of 49 and so HCHAR prints a "1" at row 1, column $25 + 1 = 26$. The SEG\$ extracts the second "1" from "1110", ASC converts it to 49, and HCHAR prints, "1", at row 1, column $25 + 2 = 27$. The entire string is displayed this way.

When you run this program, you'll see the new frequency displayed. For example, run the program and press the "K" key. You'll see "1210",

"1100", and then "9900". Keep pressing the "K" key until the cursor is at the top of the screen and you hear the lowest tone. Although the display reads "1100", the frequency is actually "110". The problem is that the rightmost zero was left over from the "1100" that was printed before. When you display something with HCHAR or VCHAR, the characters do not automatically go away since the computer does not scroll the screen as it does with PRINT. In this case, the first "1100" printed four characters since that was the length of "1100". But the "990" printed only three characters and the fourth "0" was left over from the "1100".

One way to correct this problem is to blank out the display before writing the new one. So add this line

```
205 CALL HCHAR(1,26,32,5)
```

Line 205 will write five blanks to erase the previous frequency display before the new one is written by the loop of lines 210–230. When you run this version of the program, the last zero will be blanked out correctly.

Another enhancement you can add to this program will enable you to record the tones and play them back. The following version of the program is called DRAW #3 and includes new lines 10, 15, 25, 53, 54, 103, and 300–350.

```
10 REM DRAW #3
15 DIM TONE(1000)
20 CALL CLEAR
25 L=100
30 ROW=12
40 COL=16
50 CALL HCHAR(ROW,COL,32)
51 CALL HCHAR(ROW,COL,30)
52 CALL SOUND(-1000,110*ROW,COL/2)
53 T=T+1
54 TONE(T)=110*ROW
56 IF ERASE=0 THEN 58
57 CALL HCHAR(ROW,COL,32)
58 CALL KEY(0,K,S)
60 IF S=0 THEN 50
70 IF K=83 THEN 110
80 IF K=68 THEN 130
90 IF K=75 THEN 150
100 IF K=76 THEN 170
102 IF K=87 THEN 108
103 IF K=80 THEN 300
104 IF K=69 THEN 106
105 GOTO 50
106 ERASE=1
107 GOTO 50
108 ERASE=0
109 GOTO 50
```

```

110 IF COL=1 THEN 50
115 COL=COL-1
120 GOTO 50
130 IF COL=32 THEN 50
135 COL=COL+1
140 GOTO 50
150 IF ROW=1 THEN 50
155 ROW=ROW-1
157 GOSUB 200
160 GOTO 50
170 IF ROW=24 THEN 50
175 ROW=ROW+1
180 GOSUB 200
190 GOTO 50
200 ROWSOUND$=STR$(110*ROW)
205 CALL HCHAR(1,26,32,5)
210 FOR I=1 TO LEN(ROWSOUND$)
220 CALL HCHAR(1,25+I,ASC(SEG$(ROWSOUND$,I,1)))
230 NEXT I
240 RETURN
300 FOR I=1 TO T
310 CALL SOUND(-1000,TONE(I),COL/2)
320 FOR J=1 TO L
330 NEXT J
340 NEXT I
350 GOTO 50

```

Line 15 defines the dimensioned variable TONE to store the frequencies of the tones. Line 25 defines a variable called L whose value gives a time delay when the notes are played back. Try L=1 and L=50, and you'll hear the difference. Also, be sure line 52 has a duration of -1000, as shown. Line 53 increments the subscript of TONE. Line 54 assigns the new frequency to TONE. Line 103 checks if the playback key, "P", was pressed. If so, the computer goes to line 300 and starts playing back the tones with the loop of 300-340. The duration of -1000 in the CALL SOUND statements was chosen because the tones sound musical. You can pick any other values you want. Lines 320-330 introduce a delay before the computer goes back to the CALL SOUND of line 310.

When you run this program, press the "K" key and then the "L" key to produce some tones. Then press the playback key "P". You'll notice that during playback, the cursor does not blink. You may hear a long continuous tone before the ones you recorded depending on how long you waited before pressing the "K" or "L" keys after you entered RUN. Since the computer executes lines 53 and 54 much faster when you are not changing rows, more of these will be stored. In fact, every time the cursor blinks, one frequency has been stored in TONE.

There are several ways you might like to enhance this program yourself:

1. Add a key to exit the program rather than just relying on FCTN and CLEAR.
2. Make a user-friendly display of the commands always visible at the top of the screen. You'll have to restrict the cursor movement so that it doesn't erase the command area.
3. Give the user the option of the frequencies allowed. You could even assign musical tone frequencies such as "A", "B", "C", etc. to rows. See the Appendix of Musical Tone Frequencies for the frequencies of musical notes. Just store the appropriate frequencies in an array and use ROW as a subscript.
4. Add commands to store and to retrieve the tones to tape or disk.
5. Store the intensity of the tones by saving COL in a dimensioned array. Then you can play back both the frequency as well as volume.
6. Store the row and column number of each tone and show the cursor retracing its path on the screen as the tones are played.

Chapter 12

Let's Get Graphic

Your computer has some really neat graphics features that will allow you to create great displays. In this chapter, you'll see how to use these features. We'll also show how to use computer animation in an educational game that teaches letter recognition.

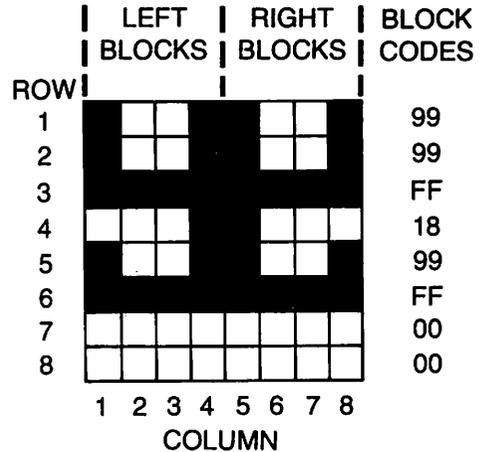
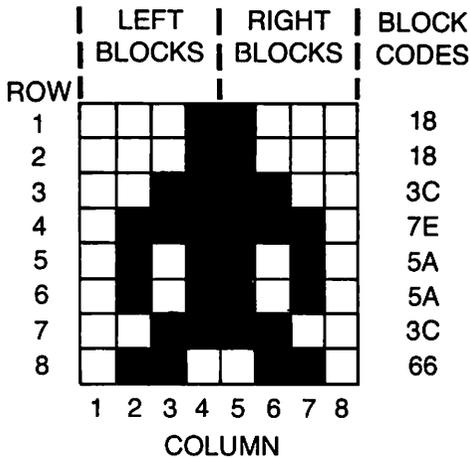
What a Character

The first feature we'll look at enables you to define your own graphics character. Instead of being limited by the ASCII set, you can create your own. To do this, just use **CALL CHAR**. As an example of how this works, enter and run the following program.

```
10 REM 2 PICTURE JUMPER
20 CALL CLEAR
30 CALL CHAR(128,"18183C7E5A5A3C66")
40 CALL CHAR(129,"9999FF1899FF0000")
50 CALL HCHAR(12,16,128)
60 FOR I=1 TO 100
70 NEXT I
80 CALL HCHAR(12,16,129)
90 FOR I=1 TO 100
100 NEXT I
110 GOTO 50
```

When you run this program, you'll see a little person jumping up and down on the screen. Line 30 is the statement which defines the character 128 as the person with arms and legs down. Line 40 defines character 129 as the person with arms and legs up. Line 50 prints character 128 on the screen at row 12, column 16. Lines 60–70 provide a short time delay before line 80 prints the person with arms and legs up. After another time delay, line 110 goes back to repeat the process.

The succession of these two characters produce **computer animation**. That is, the computer appears to make a character move. This is the same principle used in movies and TV to give the appearance of motion. When you see a series of pictures rapidly with the subject in slightly different positions, your eye interprets this as motion. The smaller the changes in the pictures, the more smooth is the illusion of motion. In our program there are only two images of the character, so the motion could be



Character assigned to Code 128.

Character assigned to Code 129.

Fig. 12-1 Pictures of the Jumper

improved by having more images. However, you would also have to show the pictures faster.

Fig. 12-1 shows a diagram of the individual **picture elements** that make up the pictures. These picture elements are called **pixels** and are represented by the blocks. Each picture is composed of 8 rows × 8 columns of pixels. The CALL CHAR tells the computer which pixels to turn on, as shown by the black blocks, and which to leave off, as shown by the white blocks.

The codes shown for the left blocks and right blocks are shown next to each picture. These codes are based on the **hexadecimal** number system. This is also called the hex system. The term hexadecimal means sixteen, so there are 16 symbols in that number system. Fig 12-2 shows the hexadecimal codes for each possible pattern of 4 blocks. The advantage of hexadecimal is that only a single symbol is required to represent any of the 16 possible rows. The decimal system would require two digits to represent the last 6 rows (rows 10, 11, 12, 13, 14, and 15).

The block codes for the left half of each figure are written to the left of the block code for the right half of each figure. For example, the top row of the character 128 has the hexadecimal code 18. The "1" is the code for the left half and the "8" is the code for the right half of the figure. The codes are written together starting from the top to the bottom row to form the entire code string for the 8 × 8 picture. Normally, there will be 16 hex codes for each picture. However, if you leave off any codes at the end, the computer interprets the missing codes as zeroes, by default. So the codes for 129 can be written as either

```
CALL CHAR(129,"9999FF1899FF0000")
```

or

```
CALL CHAR(129,"9999FF1899FF")
```

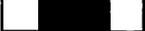
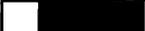
BLOCK PATTERNS	HEXADECIMAL CODE	DECIMAL NUMBER
	0	0
	1	1
	2	2
	3	3
	4	4
	5	5
	6	6
	7	7
	8	8
	9	9
	A	10
	B	11
	C	12
	D	13
	E	14
	F	15

Fig. 12-2 Hexadecimal codes for block patterns.

To show you how an additional picture can smooth the motion, let's add the picture shown in Fig. 12-3. This picture is meant to show how the person looks between the down picture and up picture. This picture will be assigned code 129 and the original 129 will be assigned code 130. The following program uses all three pictures to make a more natural-appearing jump. Notice that the program has been rewritten in terms of a FOR-NEXT loop. This makes it easier to write the program. You can also see why the intermediate picture of the jump had to be assigned code 129. Since its motion falls between the other two pictures, its code must also fall between the other two. Also, note that the delay has been reduced from 100 to 60. Since three pictures are being shown, less delay is needed.

```

10 REM 3 PICTURE JUMPER
20 CALL CLEAR
30 CALL CHAR(128,"18183C7E5A5A3C66")
40 CALL CHAR(130,"9999FF1899FF0000")
50 CALL CHAR(129,"1818FF1818FF0000")
60 FOR I=0 TO 2
70 CALL HCHAR(12,16,128+I)
80 FOR J=1 TO 60
90 NEXT J
100 NEXT I
110 GOTO 60

```

ROW									BLOCK CODES
1									18
2									18
3									FF
4									18
5									18
6									FF
7									00
8									00
	1	2	3	4	5	6	7	8	
	COLUMN								

Character assigned to Code 129 after original 129 was assigned to 130.

Fig. 12-3 — Additional Picture of the Jumper

You may wish to develop a game based on JUMPER. For example, move an object such as a cursor block towards JUMPER. If the object's position and JUMPER's are the same, then the game ends. Allow the player to use keyboard input with CALL KEY to make JUMPER jump over the object as it passes beneath. Give points for every object that JUMPER manages to jump over.

You can define your own characters for codes 128–159. You can also redefine the characters for codes 32–127. When you run the program with a redefined character, all those characters will be the ones you specify. If you have a breakpoint in your program, codes 32–127 are reset to their original characters while codes 128–159 are left alone. After the program ends or is halted by an error, any redefined characters are set back to their original while the characters for codes 128–159 are not defined. That is, codes 128–159 could be anything.

R Walk

As an example of an educational game, let's look at a program designed to teach kids the alphabet. Another application of this program is in improving touch typing. However, first enter a NEW command to delete any other program. Then enter the following lines and run. These lines are taken out of the main program and control the character animation and sound effects.

```

20 CALL CLEAR
110 CALL CHAR(128,"FC8282FC84828181")
120 CALL CHAR(129,"FC8282FC88888888")
130 CALL CHAR(130,"FC8282FC8890A0C0")
140 CALL CHAR(159,"FF090909191926C0")
150 ROW=12
    
```

```

160 MAXROW=24
170 LENGTH=20
180 LETTER=INT(26*RND)+65
190 CALL HCHAR(ROW+1,4,32,LENGTH+1)
200 CALL HCHAR(ROW+1,4,LETTER,LENGTH)
210 FOR COL=4 TO LENGTH+3
280 CALL SOUND(-1000,20*COL+110,30-COL)
290 CALL HCHAR(ROW,COL-1,32)
300 FOR J=1 TO 3
310 CALL HCHAR(ROW,COL,J+127)
320 NEXT J
530 NEXT COL
540 CALL HCHAR(ROW,COL-1,32)
550 FOR I=ROW+1 TO MAXROW
560 CALL SOUND(-1000,110*(MAXROW-I+1),0)
570 CALL HCHAR(I-1,COL,32)
580 CALL HCHAR(I,COL,159)
590 NEXT I
600 CALL SOUND(-1000,-6,0)
610 CALL HCHAR(MAXROW,COL,32)
620 CALL HCHAR(MAXROW-1,COL,159)
630 CALL HCHAR(MAXROW-1,COL,159)
640 CALL HCHAR(MAXROW-1,COL,32)
650 CALL HCHAR(MAXROW,COL,159)
660 MAXROW=MAXROW-1
670 IF MAXROW=ROW THEN 680 ELSE 210
680 END

```

When you run this program, you'll see the letter "R" walking across a row of letters. After the "R" walks over the last letter, it will fall to the bottom of the screen (row 24) and bounce up one row. "R"'s will keep walking across the letters and the "R"'s will keep piling up until they're on a level with the row. At this point the program is over.

Lines 110–130 define the characters which show the "R" walking. You may wish to change these characters to show also the rear leg of the "R" moving. Line 140 is the character code for an "R" on its side. This code is used to show the falling "R".

Line 150 defines a variable, ROW, which specifies the row that the "R" is in. Line 160 defines the maximum number of rows on the screen in the variable MAXROW. If you decrease MAXROW, the player will have fewer chances to keep playing because the pile of "R"'s will reach the row of letters sooner. Line 170 defines the variable LENGTH as how many letters the R walks over. Initially LENGTH=20. In the actual game, the LENGTH is decreased by one as the game progresses. This makes it harder for the player.

Line 180 randomly picks a number from 65–90 and assigns this to LETTER. Codes in this range represent the ASCII codes of the letters A-Z. Line 190 blanks out the previous row of letters. During the game play, the

last letter is removed. That's why the blanks are written up to LENGTH + 1. Line 200 writes the new row of letters.

Lines 210–530 are the FOR-NEXT loop that makes the "R" walk across the row. Line 280 is a sound effect that occurs every time the "R" walks over another letter. Lines 290–320 provide the computer animation of the "R" walking. The "R" is printed at a new column and these lines blank out the old "R" and make the new "R" moves its leg.

Lines 540–590 provide the visual and sound effects of the "R" falling. Lines 600–650 provide the sound effects and the appearance of the "R" bouncing. Line 660 subtracts one from the maximum row the "R" can hit. If the MAXROW equals the "R" row in line 670, the game ends by going to line 680.

The complete program is shown following. When you see a row of letters, press the same key. The changing number display shows how many points you get. When the "R" starts from the left, you can get 20 points. If you press the right key then you get the points shown on the display. Depending on where the computer is in the program, you may have to hold down the key for a second or two to get a response. Add lines 10, 30–100, 220–270, 330–520, and 690–760.

```

10 REM R WALK
20 CALL CLEAR
30 DIFFICULTY=2
40 RANDOMIZE
50 X=9
60 Y=7
70 DELAY=0
80 STRING$="SCORE=0"
90 GOSUB 700
100 CORRECT$="YOU GOT IT!"
110 CALL CHAR(128,"FC8282FC84828181")
120 CALL CHAR(129,"FC8282FC88888888")
130 CALL CHAR(130,"FC8282FC8890A0C0")
140 CALL CHAR(159,"FF090909191926C0")
150 ROW=12
160 MAXROW=24
170 LENGTH=20
180 LETTER=INT(26*RND)+65
190 CALL HCHAR(ROW+1,4,32,LENGTH+1)
200 CALL HCHAR(ROW+1,4,LETTER,LENGTH)
210 FOR COL=4 TO LENGTH+3
220 COLSTRING$=STR$(24-COL)
230 X=10
240 Y=13
250 DELAY=0
260 STRING$=COLSTRING$
270 GOSUB 690
280 CALL SOUND(-1000,20*COL+110,30-COL)
290 CALL HCHAR(ROW,COL-1,32)

```

```
300 FOR J=1 TO 3
310 CALL HCHAR(ROW,COL,J+127)
320 NEXT J
330 CALL KEY(0,K,S)
340 IF K<>LETTER THEN 530
350 X=9
360 Y=13
370 DELAY=0
380 SCORE=SCORE+24-COL
390 STRING$=STR$(SCORE)
400 CALL SOUND(200,300,3)
410 CALL SOUND(600,800,0)
420 GOSUB 700
430 X=8
440 Y=7
450 DELAY=1
460 STRING$=CORRECT$
470 GOSUB 700
480 CALL HCHAR(8,7,32,15)
490 CALL HCHAR(ROW,COL,32)
500 IF LENGTH=DIFFICULTY THEN 180
510 LENGTH=LENGTH-1
520 GOTO 180
530 NEXT COL
540 CALL HCHAR(ROW,COL-1,32)
550 FOR I=ROW+1 TO MAXROW
560 CALL SOUND(-1000,110*(MAXROW-I+1),0)
570 CALL HCHAR(I-1,COL,32)
580 CALL HCHAR(I,COL,159)
590 NEXT I
600 CALL SOUND(-1000,-6,0)
610 CALL HCHAR(MAXROW,COL,32)
620 CALL HCHAR(MAXROW-1,COL,159)
630 CALL HCHAR(MAXROW-1,COL,159)
640 CALL HCHAR(MAXROW-1,COL,32)
650 CALL HCHAR(MAXROW,COL,159)
660 MAXROW=MAXROW-1
670 IF MAXROW=ROW THEN 680 ELSE 210
680 END
690 CALL HCHAR(X,Y+1,32,5)
700 FOR I=1 TO LEN(STRING$)
710 CALL HCHAR(X,Y+I,ASC(SEG$(STRING$,I,1)))
720 NEXT I
730 IF DELAY=0 THEN 760
740 FOR I=1 TO 400
750 NEXT I
760 RETURN
```

The row of letters keeps decreasing until `LENGTH=DIFFICULTY`. The parameter `DIFFICULTY` determines how many letters will be left. Line 30 sets `DIFFICULTY=2`, so two letters will be left after you play the game a while. You may wish to increase `DIFFICULTY` to 5 or 6 for small children. Although the response speed of the computer to a keypress takes a second or two, a touch typist will score better at this game than someone who types by "hunt and peck". So this game does encourage people to learn touch typing.

When you play this game, you'll notice that if "R"'s fall and there is no "R" below, they appear to bounce from mid-air. This occurs because `MAXROW` determines how far the "R" will fall. The "R" does not have to hit another object to make it bounce. In the next chapter, we'll add some additional statements to add color, and some characters for the "R"'s to bounce against. Be sure to save `R WALK`.

Change Your Screen

You can easily change the **screen color** with the **CALL SCREEN** subprogram. Enter and run the following new program.

```

10 REM SCREEN COLOR CHANGE
20 CALL CLEAR
30 FOR I=1 TO 16
40 PRINT I
50 CALL SCREEN(I)
60 FOR J=1 TO 1000
70 NEXT J
80 NEXT I
90 GOTO 20

```

When you run this program, a number appears and you'll see the screen color for that number. This number is used as the argument of `CALL SCREEN` in line 50. The following table lists the colors and the argument of `CALL SCREEN` which specifies a color.

<u>Color-code</u>	<u>Color</u>
1	Transparent
2	Black
3	Medium Green
4	Light Green
5	Dark Blue
6	Light Blue
7	Dark Red
8	Cyan
10	Light Red
11	Dark Yellow
12	Light Yellow

13	Dark Green
14	Magenta
15	Gray
16	White

When a program is running, the standard screen color is light green, which corresponds to color-code 4. The color code can be a numeric expression, or variable.

Add Some More Color

You can make even more colorful displays if you use the CALL COLOR subprograms along with CALL SCREEN. The **CALL COLOR** subprogram allows you to specify the color of characters on the screen. To see some examples of CALL COLOR enter the following program.

```

10 REM COLOR
20 CALL CLEAR
30 INPUT "CHAR-SET, FOREGROUND, BACKGROUND=?": S, F, B
40 CALL COLOR(S, F, B)
50 FOR I=32 TO 127
60 PRINT CHR$(I);
70 NEXT I
80 GOTO 30

```

When you run this program, input 5, 16, 5. You'll see the printable character set with ASCII codes 32 to 127 displayed on the screen. Then you'll see the characters

```
@ABCDEFGG
```

in the set displayed as white characters on a blue background. Each character is shown within an individual square. The color of the characters is called the **foreground color** while the color of the rest of the square is called the **background color**.

The first argument of CALL COLOR specifies the group of characters which will be colored. The following table lists the character codes for these groups. You'll notice that @, A, B, C, D, E, F, G are in group 5 if you look up their ASCII codes from the Appendix of ASCII Character Codes.

<u>Set Number</u>	<u>Character Codes</u>
1	32-39
2	40-47
3	48-55
4	56-63
5	64-71
6	72-79
7	80-87
8	88-95
9	96-103

10	104–111
11	112–119
12	120–127
13	128–135
14	136–143
15	144–151
16	152–159

Now input the numbers 6, 16, 5. You'll see additional letters H, I, J, K, L, M, N, O also displayed as white on a blue background. Now enter 5, 16, 1. The characters in the first set will change to white letters over the background color of the screen. That's because the **transparent** color code of 1 was given as the last argument. The transparent code for the background lets the background of the screen be the background color of the character.

Try some more combinations of input with this program. Also, the Appendix of High-Resolution Color Combinations lists some of the high resolution color combinations for foreground and background. These are the easiest for a person to see. When you experiment, you'll notice that some colors are not as visible as others. For example, the combination 5, 6, 7 is very hard to see.

If you're curious as to what all the combinations look like, the following program tries all combinations of character codes, and screen colors. It also displays the screen, foreground and background codes during all these color changes. However, if you run the program as it is, it will take a while before you can see the numbers for screen, foreground and background colors. These are being printed in the lower left corner of the screen, in the region that looks like a flickering "L". To make them visible more quickly, stop the program with the FCTN and CLEAR keys. Then change line 80 to

```
80 FOR I=3 TO 16
```

so that you'll skip the screen color codes 1 and 2, and run the program again.

```
10 REM COLOR CHANGE
20 CALL CLEAR
30 FOR I=32 TO 127
40 PRINT CHR$(I);
50 NEXT I
60 PRINT : "SCREEN=" : "FOREGROUND=" : "BACKGROUND="
70 FOR L=1 TO 12
80 FOR I=1 TO 16
90 CALL SCREEN(I)
100 R=1
110 STRING$=STR$(I)
120 GOSUB 270
130 FOR J=1 TO 16
140 FOR K=1 TO 16
```

```

150 CALL COLOR(L,J,K)
160 R=2
170 STRING$=STR$(J)
180 GOSUB 270
190 R=3
200 STRING$=STR$(K)
210 GOSUB 270
220 NEXT K
230 NEXT J
240 NEXT I
250 NEXT L
260 END
270 CALL HCHAR(20+R,14,32,3)
280 FOR M=1 TO LEN(STRING$)
290 CALL HCHAR(20+R,M+14,ASC(SEG$(STRING$,M,1)))
300 NEXT M
310 RETURN

```

Now that you've seen how colorful your computer can be, let's add some color and rocks to the R WALK Program. First, let's add the rocks with lines 172–176 below.

```

172 FOR I=ROW+2 TO 24
174 CALL HCHAR(I,4,42,LENGTH)
176 NEXT I

```

This loop uses asterisks, code 42, to simulate rocks.

Now let's delete a column of asterisks whenever the "R" falls. However, we'll always have one more asterisk than MAXROW as the object that the "R" will bounce from.

```

205 CALL VCHAR(ROW+2,LENGTH+4,32,MAXROW-ROW-1)

```

Enter and run the program for the above lines. Now add the following lines to give color.

```

41 CALL SCREEN(13)
42 FOR I=5 TO 8
43 CALL COLOR(I,16,5)
44 NEXT I
45 CALL COLOR(2,7,12)
46 CALL COLOR(13,16,1)

```

Line 41 makes the screen dark green. Lines 42–44 make all the letters of the alphabet white on dark blue. Since the letters are in different groups, you need a separate CALL COLOR statement for each group. Line 45 makes the asterisks appear dark red on light yellow, while line 46 makes the "R" white on transparent.

What's That Character?

In some programs, you'll find it useful to read a character from anywhere on the display screen. For example, in some game programs you'll want to find out if two objects, such as a spaceship and a meteor, collide. This can be done by keeping track of the positions of the two objects. Another way is to use the **CALL GCHAR** subprogram to read the value of a character on the screen. For example, enter and run the following program.

```
10 CALL CLEAR
20 CALL HCHAR(12,16,42)
30 CALL GCHAR(12,16,CHAR)
40 PRINT CHAR,CHR$(CHAR)
```

You'll see an asterisk printed. Line 20 will print an asterisk, ASCII code 42, on the screen at row 12, column 16. Line 30 will read the ASCII code of the character displayed at that location. Actually, GCHAR will read the value from that portion of the computer memory, called the **display memory**, which stores characters for the display. A certain display memory location contains the value of the character stored in row 12, column 16. The CALL GCHAR reads that location to find the value of the character and assigns it to the variable used as the third argument of GCHAR. The first and second arguments of GCHAR are the row and column which GCHAR will read. Any variable name can be used as the third argument. Likewise, the CALL HCHAR and CALL VCHAR subprograms can put the value into a display memory location.

Alpha Pilot

You are an Alpha Pilot assigned to the Space Shuttle for special missions. Your assignment is to retrieve satellites that need repair. There are twenty-six satellites labeled by A to Z that you want to retrieve in the following game. This game gives a good illustration of how GCHAR is useful in a **dynamic** game. The term dynamic means changing. In this game, the game display changes all the time. In contrast, the number guessing game with "High" and "Low" is an example of a **static** display. The term static means unchanging. Except for when you enter input, the game display did not change. In this Alpha Pilot game, you move your ship with the "S" key for left and the "D" key for right. Just touch the nose of your ship to a satellite. You'll hear the recognition sounds and be awarded a point. Your total score will be printed in the upper right corner of the screen.

But watch out. There are meteors everywhere and when one hits the front of your ship, the game is over. However, you can move sideways into a meteor because your disintegrator shields will protect your ship from the side. The area to watch out for is the two display positions directly in front or to the side of the ship. The ship is two columns wide and a hit in either column will end you. Likewise, you can capture a satellite in those regions.

In order to see these two display positions, you may want to add the lines

```
275 CALL HCHAR(9,COL,63)
315 CALL HCHAR(9,COL+1,63).
```

When you run the game with lines 275 and 315, you'll see two question marks printed in front of the ship. These "?" marks always appear in the direction that the ship is traveling. Move the spaceship back and forth with the "S" and "D" keys and watch how these positions move. You retrieve a satellite when one touches these regions. Enter and run the following program. Good hunting.

```
10 REM ALPHA PILOT
20 RANDOMIZE
30 CALL CHAR(128,"1FFFFFFFFF7F3F1F")
40 CALL CHAR(129,"F8FFFFFFFFFEFCF8")
50 CALL CHAR(130,"0F07070707070303")
60 CALL CHAR(131,"F0E0E0E0E0E0C0C0")
70 CALL CHAR(132,"0303030303010101")
80 CALL CHAR(133,"C0C0C0C0C0808080")
90 CALL CHAR(136,"183C7EFFFF7E3C18")
100 CALL COLOR(14,7,1)
110 CALL COLOR(13,6,1)
120 FREQ=20
130 MAXCOUNT=26*FREQ+16
140 FOR I=3 TO 8
150 CALL COLOR(I,2,13)
160 NEXT I
170 CALL CLEAR
180 CALL SCREEN(2)
190 ROW=6
200 COL=16
210 COUNT=0
220 LETTER=0
230 SCORE=0
240 OLDCOL=COL
250 COUNT=COUNT+1
260 IF COUNT=MAXCOUNT THEN 660
270 CALL GCHAR(9,COL,MET)
280 IF MET=136 THEN 660
290 IF MET<65 THEN 310
300 GOSUB 700
310 CALL GCHAR(9,COL+1,MET)
320 IF MET=136 THEN 660
330 METCOL=INT(20*RND)+4
340 IF MET<65 THEN 360
350 GOSUB 700
360 IF COUNT/FREQ<>INT(COUNT/FREQ) THEN 420
```

```

370 IF LETTER=90 THEN 420
380 LETTER=COUNT/FREQ+64
390 PRINT TAB(METCOL);CHR$(LETTER)
400 CALL SOUND(100,30*METCOL,3)
410 GOTO 430
420 PRINT TAB(METCOL);CHR$(136)
430 CALL SOUND(-100,30*METCOL,3)
440 CALL HCHAR(5,OLDCOL-2,32,5)
450 CALL VCHAR(5,OLDCOL,32,3)
460 CALL HCHAR(6,COL,128)
470 CALL HCHAR(6,COL+1,129)
480 CALL HCHAR(7,COL,130)
490 CALL HCHAR(7,COL+1,131)
500 CALL HCHAR(8,COL,132)
510 CALL HCHAR(8,COL+1,133)
520 CALL KEY(0,K,S)
530 IF S=0 THEN 250
540 IF K=83 THEN 580
550 IF K=68 THEN 620
560 OLDCOL=COL
570 GOTO 250
580 IF COL=4 THEN 250
590 COL=COL-1
600 OLDCOL=COL+2
610 GOTO 250
620 IF COL=24 THEN 250
630 COL=COL+1
640 OLDCOL=COL-1
650 GOTO 250
660 PRINT "FINAL SCORE=";SCORE
670 INPUT "ANOTHER GAME? (Y OR N)":ANSWERS
680 IF ANSWERS="Y" THEN 170
690 END
700 SET=INT((MET-32)/8)+1
710 CALL SOUND(-1000,440,3)
720 FOR I=1 TO 6
730 FOR J=1 TO 10
740 NEXT J
750 CALL COLOR(SET,2,5)
760 FOR J=1 TO 10
770 NEXT J
780 CALL COLOR(SET,2,13)
790 NEXT I
800 CALL SOUND(1500,784,0)
810 SCORE=SCORE+1
820 CALL HCHAR(3,21,32,2)
830 STRING$=STR$(SCORE)

```

```

840 FOR I=1 TO LEN(STRING$)
850 CALL HCHAR(4,20+I,ASC(SEG$(STRING$,I,1)))
860 NEXT I
870 RETURN

```

Lines 30–80 define the characters used for the Space Shuttle. The Shuttle is defined as six characters in three rows and two columns. Lines 30 and 40 define the left and right rear, lines 50 and 60 define the left and right middle, while lines 70 and 80 are the left and right front. Line 90 defines the meteors. Lines 100 and 110 are the color of the meteors and Shuttle, respectively.

Line 120 defines a parameter that determines how often a letter will appear. In this case, the frequency of occurrence, `FREQ`, is set so that a letter will appear every 20 rows. If you decrease `FREQ`, the game becomes harder because you have less time to pilot your spaceship to a letter. Line 130 defines a constant which determines how many `PRINT` statements the program will make before it ends automatically. Since there are 26 letters in the alphabet, the product of `26*FREQ` tells how many `PRINT`'s are needed to make all of them appear on the bottom of the screen. The extra 16 is to allow the last Z to reach the spaceship. Once the last Z has reached the spaceship, you can't get any more points and the game ends.

Lines 140–160 set up the colors of the letters and numbers in the program. Lines 190–200 define the initial position of the left rear of the spaceship. Line 210 defines a variable which counts how many `PRINT` statements have been executed. When `COUNT = MAXCOUNT` in line 260, the program goes to 660 to end the game. Line 220 initializes the variable `LETTER` which contains the ASCII code of the latest letter which has been printed or will be printed. Line 230 defines the variable `SCORE`, which contains the score. Each time you pick up a letter, `SCORE` increases by 1 in line 810. Lines 820–860 blank out the old score and print the new score.

Line 270 checks the square one row in front of the spaceship to the left while line 310 checks one row and to the right of the ship. Since the spaceship is composed of two columns, lines 270 and 310 check in front of these columns to see if a letter or meteor is present.

Lines 280 and 320 go to 660 to end the game if a meteor is detected by the `GCHAR`'s. Line 330 calculates the random column position of the next meteor that will appear as an integer from 4 to 23. If the character is not a meteor and not a letter, line 340 goes to 360.

Line 360 checks if it's time to print a letter instead of a meteor. For example, if `COUNT = 40` and `FREQ = 20`, then the `IF` test of line 360 is true and the computer goes to 420 to print a meteor. However, if `COUNT` is not a multiple of 20, the test fails and the computer goes to line 370 after 360.

Line 370 checks if the letter "Z" has been printed. If so, the computer goes to 420 and does not print any more letters. Try this with `FREQ = 1`. Only letters will be printed at first and then only meteors. Notice that the game ends when the "Z" is on the same row as the nose of the spaceship.

If the computer gets to line 350, it must have found a character greater than 65 for MET. Since it can't be a meteor, it must be a letter. The computer then goes to the subroutine at line 700. Line 700 first determines the character set that the letter is in. Note that LETTER contains the code of the last letter that was printed or will be printed on the bottom line of the display. That's why MET is used in line 700. For example, if an "A" is detected, then SET = 5. Line 710 produces the first note of a sound effect while line 800 produces the second note. These CALL SOUND statements were separated by the loop of 720–790 to allow time for their tones to be heard and not to delay the program. The loop of 720–790 produces color effects by changing the color of the character set of the letter that was detected by the ship.

Line 380 calculates the ASCII code of the next letter to be printed. For example, if COUNT = 40 and FREQ = 20 then LETTER = 40/20 + 64 = 66. So the letter "B" is printed. Line 390 prints the letter and 400 produces a sound effect. Line 420 prints a meteor and 430 produces a sound effect when the meteor is printed. Lines 440–450 erase part of the old image of the spaceship that won't be erased by the new printing of the spaceship. Lines 460–510 print the spaceship. Note that in order to reduce the printing time, no FOR-NEXT loop was used to print the spaceship. Although it saves you typing and computer memory, it does take the computer more time to execute a loop. You'll notice that the ship is still printed rather slowly, so efforts were made to reduce printing time. As shown in an earlier chapter, another way to squeeze a little more speed out of a program is to use variable names for constants whenever possible. By using a variable, the computer does not have to convert a number to the internal binary form every time the line is executed. This can be very helpful if you're dealing with numbers that have a lot of digits. Also, you can use shorter variable names since the computer will then take less time to recognize a name. Lines 520–650 move the ship horizontally and should be familiar from the drawing programs.

This type of program really pushes the standard BASIC in your console to its limits. You'll notice that the spaceship does not move as smoothly as an arcade game. That's because the arcade game is usually written in **machine language**, or has special hardware. Machine language is called a low-level language. BASIC is called a high-level language because its vocabulary of PRINT, RUN, etc. match English language words you're familiar with. In contrast, machine level programming is designed for the convenience of the particular computer. If you'd like more information on these subjects see the book, *Foundations of Computer Technology*.

All of the BASIC commands and statements that you enter are first processed by a program in the computer. This is called an **interpreter** and it translates every command and program line into machine code when it is encountered. In fact, that's why a BASIC program is much slower than a machine language program.

The BASIC interpreter converts every line of BASIC to machine code only when it executes that line. In contrast, all of the machine code is already in the language that the computer can understand. By skipping the interpretation step, the computer can operate hundreds of times faster. That's why arcade games are so much faster than BASIC games. However, it is much harder for most people to write and to debug machine language programs. For this reason, a higher-level language called **assembly language** was developed. There is a specific machine language and its assembly language for every computer. A program called an assembler converts assembly language statements into machine code.

Another alternative to faster programs is a **compiler**. A compiler is a program which converts all the statements into machine language. This skips the interpreter, so a compiled program is much faster than an interpreted one. The most common compilers are available for languages like FORTRAN, COBOL, and C. However, BASIC compilers are also available for some brands of computers. Although a compiler speeds up execution, it does not help you debug a program. In fact, it's just the opposite. A standard compiler can't let you set breakpoints, print out variables, trace lines, etc. This is because when the program is converted to machine language, all the information about line numbers is lost.

You can achieve faster execution of programs with the Extended BASIC cartridge from Texas Instruments. This cartridge plugs into your computer and adds a number of new features besides increasing execution speed. For example, the cartridge lets you use **sprites** in games. A sprite is a graphic image that is under the control of special hardware. For example, the spaceship in Alpha Pilot could be defined as a sprite. The sprite moves smoothly under control of the hardware so that the computer does not have to keep executing statements to move it. Sprites also can be easily magnified using an extended BASIC command. So a single sprite could be used instead of the 6 picture elements used in the game. Sprites can be defined in 32 different layers called **planes**. A sprite in one layer can pass in front or behind one in another layer. This feature gives much more realistic effects. The extended BASIC also has more commands for use with disks. For example, you can merge program lines from different programs. Other BASIC commands that we did not discuss, such as **BYE, DELETE** and **EOF** are used with disks. Consult your TI-99/4A User's Reference Guide and disk manual for more details.

Illusions

You can achieve some pretty amazing effects with your TI-99/A if you use its built in features. For example, suppose you want to develop a game in which the background shows waves moving down the screen. The game could consist of steering a ship over the waves and avoiding obstacles such as rocks and sharks.

There are several ways you can make the waves move down the screen without using CALL HCHAR and CALL VCHAR to move the waves. The

disadvantage of these subprograms is that it takes the computer a long time to move many objects. For example, enter and run the following new program.

```

10 REM WAVES 1
20 CALL CLEAR
30 CALL CHAR(128,"FFFFFFFF")
40 CALL CHAR(136,"FFFFFFFF")
50 FOR I=0 TO 22 STEP 2
60 CALL HCHAR(1+I,1,128,32)
70 CALL HCHAR(2+I,1,136,32)
80 NEXT I
90 CALL COLOR(13,16,16)
100 CALL COLOR(14,6,6)
110 CALL COLOR(13,6,16)
120 CALL COLOR(14,16,6)
130 CALL COLOR(13,6,6)
140 CALL COLOR(14,16,16)
150 CALL COLOR(13,16,6)
160 CALL COLOR(14,6,16)
170 GOTO 90

```

When you run the WAVES 1 program, you'll see waves moving down the screen. Just imagine that the white lines are the peaks of the waves, while the blue lines are the bottoms of the waves. Lines 30 and 40 define two special characters. Each character is half a square. In terms of appearance, each character is the same size. However, their character codes were chosen to be in different color groups. Lines 50-80 print these characters down the screen in alternating rows. Lines 90-160 change the foreground and background colors in each special character so that the waves appear to move down the screen. Changing the colors gives the illusion of movement.

Notice that lines 90-160 use the CALL COLOR command to change the colors of all the special characters at once. This changes the colors pretty quickly. However, if you observe the waves closely, you'll see the blue and white regions get bigger and smaller. Theoretically, the width of each blue and white line should remain the same. However, the standard BASIC in your console is not quite fast enough to eliminate this effect when the white is printed over the blue and the blue is printed over the white.

There is a way to speed things up. Enter and run the following new version of WAVES

```

10 REM WAVES 2
20 CALL CLEAR
30 N=1
40 CALL CHAR(128,"FFFFFFFF")
50 CALL HCHAR(1,1,128,768)
60 CALL KEY(0,K,S)
70 IF K<>75 THEN 100

```

```

80 N=2*N
90 GOTO 120
100 IF K<>76 THEN 120
110 N=N/2
120 FOR I=1 TO N
130 NEXT I
140 CALL COLOR(13,6,16)
150 FOR I=1 TO N
160 NEXT I
170 CALL COLOR(13,16,6)
180 GOTO 60

```

When you run this program, you'll see the blue and white lines as straight. The lines do not get bigger and smaller as in the WAVES 1 program. If you press the "K" key, you can slow down the waves, while the "L" key speeds them up.

Line 40 defines a single special character. Line 50 fills the screen with this character. Lines 60-110 accept input from the keyboard to determine N. The variable N is used in the loop of lines 120-130 and 150-160 to delay the motion of the waves. Without some delay, the motion is extremely rapid. Lines 140 and 170 actually produce the wave effect. Line 140 sets the top half of the special character to blue and the bottom half to white. Line 170 reverses this and makes the top half white and bottom half blue.

If you have a lot of objects to move, you can speed up a game by using CALL COLOR and special characters as much as possible during the playing time. Avoid HCHAR, VCHAR and other functions as much as possible during the playing time to move many objects. You can use HCHAR and VCHAR to set up the game, and then use CALL COLOR while playing. This is the same basic idea as using PRINT in ALPHA PILOT to move all the meteors at once.

Another example of using color changes to give the illusion of movement is shown in the following program. Enter and run this new program. You'll see a 7 × 7 square in the screen with colors moving around. This square is actually a spiral, and four colors move continually from the center of the spiral to the outside.

```

10 REM COLOR SPIRAL 1
20 CALL CLEAR
30 NUMCOL=4
40 C(1)=5
50 C(2)=7
60 C(3)=11
70 C(4)=16
80 A$="FFFFFFFFFFFFFFFF"
90 CALL CHAR(128,A$)
100 CALL CHAR(136,A$)
110 CALL CHAR(144,A$)
120 CALL CHAR(152,A$)

```

```

130 N=7
140 SP=8
150 FOR ROW=1 TO N
160 FOR COL=1 TO N
170 READ CHAR
180 CHAR=128+8*(CHAR-1)
190 CALL HCHAR(SP+ROW,SP+COL,CHAR)
200 NEXT COL
210 NEXT ROW
220 FOR I=1 TO NUMCOL
230 CALL COLOR(12+I,C(I),C(I))
240 NEXT I
250 TEMP=C(NUMCOL)
260 FOR I=NUMCOL-1 TO 1 STEP -1
270 C(I+1)=C(I)
280 NEXT I
290 C(1)=TEMP
300 GOTO 220
310 DATA 1,4,3,2,1,4,3
320 DATA 2,1,4,3,2,1,2
330 DATA 3,2,1,4,3,4,1
340 DATA 4,3,2,1,2,3,4
350 DATA 1,4,3,4,1,2,3
360 DATA 2,1,2,3,4,1,2
370 DATA 3,4,1,2,3,4,1

```

The variable NUMCOL in line 30 defines the number of colors used in the spiral. The four color codes are stored in the array C in lines 40-70. Lines 90-120 generate four character codes for the spiral. The variable N in line 130 defines the number of rows in the spiral. Line 140 defines the initial starting position of the spiral as the variable SP. Actually, SP+1 represents the upper, left-hand, corner row and column number of the spiral. That is, the upper left edge of the spiral is in row 9 and column 9. The spiral is drawn tightly packed into a square and so the number of rows equals the number of columns. Lines 150-210 print the spiral on the screen using the data in lines 310-370. Each number in the DATA statements is used to generate a character. For example, if the data is a "1", then CHAR=1 from line 170 and CHAR=128 from line 180. Likewise, when line 170 reads CHAR=2, then line 180 calculates CHAR=136. If CHAR=3, then CHAR=144 from line 180 and if CHAR=4, then CHAR=152 from line 180.

The center of the spiral is in the fourth item in the DATA statement of line 340, and is the "1". Fig. 12-4 shows the data elements with arrows indicating the pattern of the spiral. The colors will appear to move along this pattern. Moving to the right by one gives a "2". Moving directly up from the "2" you'll see a "3". Move left from the "3" and you'll see a "4" and then repeat with a "1" since there are only four special characters. The spiral path is continued by moving down from the "1" to elements

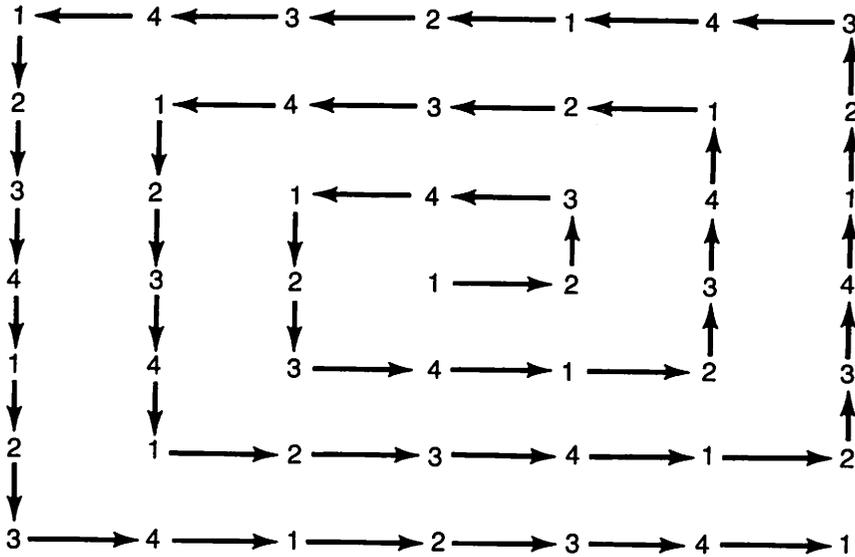


Fig. 12-4 Spiral, showing the pattern of the four special characters which make its shape.

“2” and “3” and then to the right with elements “4,” “1,” “2,” and so forth. Lines 220-240 print the colors.

Lines 250-290 permute the color elements so that the colors shift by one square. For example, the following table shows the initial values of the colors and then their values each time after lines 250-290.

<u>Initial Colors</u>	<u>Permutations of Colors</u>
C(1) = 5	16 11 7 5
C(2) = 7	5 16 11 7
C(3) = 11	7 5 16 11
C(4) = 16	11 7 5 16

From the table above, you can see that the last permutation has given back the original color sequence. These permutations keep repeating over and over again so that the colors keep moving. You can speed up the color changes a little by directly entering sixteen appropriate CALL COLOR statements instead of lines 220-290.

There are several ways you can increase the size of the color spiral. The simplest way is to add appropriate data statements. Another way is to use FOR-NEXT loops to generate the spiral. This version of the Color Spiral Program is shown in the following listing. Rather than your having to enter DATA statements, this program uses FOR-NEXT loops to generate the spiral. Fig. 12-4 shows the pattern of the spiral and you’ll see this pattern appear on your screen as the computer makes the pattern. You’ll see a much larger spiral on the screen when you run this version compared to the first version of the program. Of course, you could add more data state-

ments to the Color Spiral 1 version to generate a larger spiral, but that will take a lot of typing. Lines 20-120 are the same as the Color Spiral 1 Program.

```

10 REM COLOR SPIRAL 2
20 CALL CLEAR
30 NUMCOL=4
40 C(1)=5
50 C(2)=7
60 C(3)=11
70 C(4)=16
80 A$="FFFFFFFFFFFFFFF"
90 CALL CHAR(128,A$)
100 CALL CHAR(136,A$)
110 CALL CHAR(144,A$)
120 CALL CHAR(152,A$)
130 ROW=12
140 COL=16
150 LENGTH=1
160 CHAR=128
170 CALL HCHAR(ROW,COL,CHAR)
180 NEWCOL=COL+LENGTH
190 FOR I=COL TO NEWCOL
200 GOSUB 550
210 CALL HCHAR(ROW,I,CHAR)
220 NEXT I
230 COL=NEWCOL
240 NEWROW=ROW-LENGTH
250 FOR I=ROW TO NEWROW STEP -1
260 GOSUB 550
270 CALL HCHAR(I,COL,CHAR)
280 NEXT I
290 ROW=NEWROW
300 LENGTH=LENGTH+1
310 NEWCOL=COL-LENGTH
320 FOR I=COL TO NEWCOL STEP -1
330 GOSUB 550
340 CALL HCHAR(ROW,I,CHAR)
350 NEXT I
360 COL=NEWCOL
370 NEWROW=ROW+LENGTH
380 FOR I=ROW TO NEWROW
390 GOSUB 550
400 CALL HCHAR(I,COL,CHAR)
410 NEXT I
420 ROW=NEWROW
430 COUNT=COUNT+1
440 LENGTH=LENGTH+1

```

```

450 IF COUNT<11 THEN 170
460 FOR I=1 TO NUMCOL
470 CALL COLOR(12+I,C(I),C(I))
480 NEXT I
490 TEMP=C(NUMCOL)
500 FOR I=NUMCOL-1 TO 1 STEP -1
510 C(I+1)=C(I)
520 NEXT I
530 C(1)=TEMP
540 GOTO 460
550 CHAR=CHAR+8
560 IF CHAR<160 THEN 580
570 CHAR=128
580 RETURN

```

Lines 20-120 are the same in this program as the previous spiral program. Lines 130 and 140 define the center row and column of the spiral. The variable LENGTH initialized in line 150 controls how many squares will be printed in the same direction as the spiral is drawn. Line 160 defines the variable CHAR which contains the character number to be printed in the spiral. Line 170 prints the center square of the spiral. Line 180 defines a variable called NEWCOL which defines the ending column of the spiral. NEWCOL is also the new initial column of the spiral.

The loop of lines 190-220 prints the part of the spiral from left to right. The GOSUB 550 subroutine increments the character by 8 and checks if the character is less than 160. If so, the program returns and prints the character. If the character code is 160, then the character is reset to 128. Line 230 sets the starting column to the ending column. Line 240 calculates what the ending row of the spiral should be as the spiral is drawn up. Lines 250-280 draw the spiral going up. Line 290 sets the starting row to the ending row position while line 300 increases the length by one and line 310 sets the ending column size. Lines 320-350 draw the spiral going from right to left. Lines 360 and 370 adjust the column and ending row while the loop of lines 380-410 draw the spiral from left to right. Line 420 adjusts the beginning row position.

The variable COUNT in line 430 counts how many times the spiral has looped around itself. Decrease the 11 in line 450 to get a smaller spiral. Lines 460-540 act the same as lines 220-300 in the Color Spiral 1 Program.

The Great Adventure

Game programs are a fun and educational way to learn about programming. If you'd like to see some more examples of game programs, then get *Timelost for the TI-99/4A*. This is a best selling, computer adventure set in comic book form about a teenage boy, his kid sister and their mysterious friend from the future as they battle the Warlord of the Universe. Program listings are included which match the action of the story. Just type in

the listings and you can battle with them. Also included are detailed explanations of the listings so that you can learn how the games operate. Another reason for including detailed listings is so that you can modify and enhance the games.

If you'd like to learn more about computer hardware, see *Foundations of Computer Technology* and *Modern Computer Concepts*. These books provide an introduction to computer history, terminals, printers, disks, memory, data communications, and a guide to purchasing a computer system. Two other books *BASIC: Fundamental Concepts* and *BASIC: Advanced Concepts* cover two other popular dialects of BASIC. One version is a BASIC made by Digital Equipment Corporation and the other is made by Microsoft, Inc. The programs in these books are shown for both types of BASIC. You'll learn how to convert programs written in one version to another and the pitfalls that can occur in converting a program written for one brand of computer to run on another. Often in magazines or books you'll see programs written for a different brand of computer that you'd like to run on your computer. However, there may be more involved than just changing the vocabulary of BASIC.

The BASIC books above also cover other topics such as plotting on a terminal, factorials, and exact precision arithmetic. For example, the TI 99/4A prints to ten significant figures, although internally it's accurate to 13 or 14. The programs in *BASIC: Advanced Concepts* allow you to add, to subtract, and to multiply numbers with 200 or more digits! The books are available at a discount when purchased in groups of two or four from the publisher, Howard W. Sams, Inc., 4300 W. 62nd Street, Indianapolis, IN 46268.

Computer magazines are a good source of material about computers. They carry games, tips, and articles about programming and many other topics. The Magazine

99'er Home Computer Magazine
P. O. Box 5537
Eugene, OR 97405
(503) 485-8796

is exclusively devoted to Texas Instruments computers and contains many articles and features about TI computers. Also included is information about LOGO, a popular computer language primarily aimed at children.

Another good magazine is

Compute! The Journal for Progressive Computing
P. O. Box 5406
Greensboro, NC 27403
(919) 275-9809

This magazine publishes programs, articles and features about the TI-99/4A and other brands of computers. The nice thing about *Compute* is that many of their program listings are given for several different brands of computers.

If you've worked through all the material in this book, you've come a long way. At this point, if you have optional accessories such as the Speech Module or joysticks, you should be able to read their documentation and to write programs using them. However, the key to becoming a good programmer is practice. You can learn to program only by programming. Pick a topic that you're interested in and write a program for it. Start with a good design and test each part of the program before continuing. If things don't work out, relax and take a break for a while. Remember, computers are fun only if you're having fun using them. Computers are the great adventure of our time. Along with millions of other people learning about computers, you're traveling the right road to that adventure.

Appendices

NOTE: Except for the Appendix of Order of Priorities of Operators, all other appendices have been reproduced from the *Texas Instruments TI-99/4A Computer User's Reference Guide*, copyrighted by Texas Instruments, Inc. This material is reproduced with the permission of Texas Instruments, Inc. and may not be reproduced without the written permission of Texas Instruments, Inc.

Appendix of Order of Priorities of Operators

Exponentiation \wedge

Negation -

Multiplication and Division $*, /$

Addition and Subtraction $+, -$

Relational Operators $>, =, <, <>, <=, >=$

Concatenation $\&$

Appendix of Reserved Words

ABS	GOTO	RES
APPEND	IF	RESEQUENCE
ASC	INPUT	RESTORE
ATN	INT	RETURN
BASE	INTERNAL	RND
BREAK	LEN	RUN
BYE	LET	SAVE
CALL	LIST	SEG\$
CHR\$	LOG	SEQUENTIAL
CLOSE	NEW	SGN
CON	NEXT	SIN
CONTINUE	NUM	SQR
COS	NUMBER	STEP
DATA	OLD	STOP
DEF	ON	STR\$
DELETE	OPEN	SUB
DIM	OPTION	TAB
DISPLAY	OUTPUT	TAN
EDIT	PERMANENT	THEN
ELSE	POS	TO
END	PRINT	TRACE
EOF	RANDOMIZE	UNBREAK
EXP	READ	UNTRACE
FIXED	REC	UPDATE
FOR	RELATIVE	VAL
GO	REM	VARIABLE
GOSUB		

Appendix

ASCII CHARACTER CODES

The defined characters on the TI-99/4A Computer are the standard ASCII characters for codes 32 through 127. The following chart lists these characters and their codes.

ASCII CODE	CHARACTER	ASCII CODE	CHARACTER	ASCII CODE	CHARACTER
32	(space)	65	A	97	A
33	! (exclamation point)	66	B	98	B
34	" (quote)	67	C	99	C
35	# (number or pound sign)	68	D	100	D
36	\$ (dollar)	69	E	101	E
37	% (percent)	70	F	102	F
38	& (ampersand)	71	G	103	G
39	' (apostrophe)	72	H	104	H
40	((open parenthesis)	73	I	105	I
41) (close parenthesis)	74	J	106	J
42	* (asterisk)	75	K	107	K
43	+ (plus)	76	L	108	L
44	, (comma)	77	M	109	M
45	- (minus)	78	N	110	N
46	. (period)	79	O	111	O
47	/ (slant)	80	P	112	P
48	0	81	Q	113	Q
49	1	82	R	114	R
50	2	83	S	115	S
51	3	84	T	116	T
52	4	85	U	117	U
53	5	86	V	118	V
54	6	87	W	119	W
55	7	88	X	120	X
56	8	89	Y	121	Y
57	9	90	Z	122	Z
58	: (colon)	91	[(open bracket)	123	{ (left brace)
59	; (semicolon)	92	\ (reverse slant)	124	
60	< (less than)	93] (close bracket)	125	} (right brace)
61	= (equals)	94	^ (exponentiation)	126	~ (tilde)
62	> (greater than)	95	_ (line)	127	DEL (appears on screen as a blank.)
63	? (question mark)	96	` (grave)		
64	@ (at sign)				

These characters are grouped into sixteen *sets* for use in color graphics programs.

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

Two additional characters are predefined on the TI-99/4A Computer. The *cursor* is assigned to ASCII code 30, and the *edge character* is assigned to code 31.

Appendix

FUNCTION AND CONTROL KEY CODES

Codes are also assigned to the function and control keys, so that these can be referenced by the CALL KEY subprogram in TI BASIC. The codes assigned depend on the *key-unit* value specified in a CALL KEY program statement.

Function Key Codes

Codes				
<i>TI-99/4 & BASIC Modes</i>	<i>Pascal Mode</i>		<i>Function Name</i>	<i>Function Key</i>
1	129		AID	FCTN 7
2	130		CLEAR	FCTN 4
3	131		DELeTe	FCTN 1
4	132		INSert	FCTN 2
5	133		QUIT	FCTN =
6	134		REDO	FCTN 8
7	135		ERASE	FCTN 3
8	136		LEFT arrow	FCTN S
9	137		RIGHT arrow	FCTN D
10	138		DOWN arrow	FCTN X
11	139		UP arrow	FCTN E
12	140		PROD'D	FCTN 6
13	141		ENTER	ENTER
14	142		BEGIN	FCTN 5
15	143		BACK	FCTN 9

Control Key Codes

Codes				
<i>BASIC Mode</i>	<i>Pascal Mode</i>	<i>Mnemonic Code</i>	<i>Press</i>	<i>Comments</i>
129	1	SOH	CONTROL A	Start of heading
130	2	STX	CONTROL B	Start of text
131	3	ETX	CONTROL C	End of text
132	4	EOT	CONTROL D	End of transmission
133	5	ENQ	CONTROL E	Enquiry
134	6	ACK	CONTROL F	Acknowledge
135	7	BEL	CONTROL G	Bell
136	8	BS	CONTROL H	Backspace
137	9	HT	CONTROL I	Horizontal tabulation
138	10	LF	CONTROL J	Line feed
139	11	VT	CONTROL K	Vertical tabulation
140	12	FF	CONTROL L	Form feed
141	13	CR	CONTROL M	Carriage return
142	14	SO	CONTROL N	Shift out
143	15	SI	CONTROL O	Shift in
144	16	DLE	CONTROL P	Data link escape
145	17	DC1	CONTROL Q	Device control 1 (X-ON)
146	18	DC2	CONTROL R	Device control 2
147	19	DC3	CONTROL S	Device control 3 (X-OFF)
148	20	DC4	CONTROL T	Device control 4
149	21	NAK	CONTROL U	Negative acknowledge
150	22	SYN	CONTROL V	Synchronous idle
151	23	ETB	CONTROL W	End of transmission block
152	24	CAN	CONTROL X	Cancel
153	25	EM	CONTROL Y	End of medium
154	26	SUB	CONTROL Z	Substitute
155	27	ESC	CONTROL .	Escape
156	28	FS	CONTROL ;	File separator
157	29	GS	CONTROL =	Group separator
158	30	RS	CONTROL 8	Record separator
159	31	US	CONTROL 9	Unit separator

Appendix

KEYBOARD MAPPING

The following diagrams illustrate the key codes returned in the four keyboard modes specified by the *key-unit* value in the CALL KEY statement. The figures on the upper key face are function codes, and the lower figures are control codes.

3 1	4 2	7 3	2 4	14 5	12 6	1 7	6 8	15 9	0	5 -
Q	W	11 E	R	T	Y	U	I	O	P	/
A	B S	9 D	F	G	H	J	K	L	;	13 ENTER
SHIFT	Z	10 X	C	V	B	N	M	,	.	SHIFT
ALPHA LOCK	CTRL	SPACE							FCTN	

Figure 1. Standard T1-99/4 Keyboard Scan.

Key unit = 3. Both upper- and lower-case alphabetical characters returned as upper-case.
Function codes = 1-15. No control characters active.

131 1	132 2	135 3	130 4	142 5	140 6	129 7	134 8 30	143 9 31	0	133 - 29
Q 17	W 23	139 E 5	R 18	T 20	Y 25	U 21	I 9	O 15	P 16	/
A 1	136 S 19	137 D 4	F 6	G 7	H 8	J 10	K 11	L 12	;	141 ENTER 28
SHIFT	Z 26	138 X 24	C 3	V 22	B 2	N 14	M 13	,	.	SHIFT
ALPHA LOCK	CTRL	SPACE							FCTN	

Figure 2. Pascal Keyboard Scan.

Key-unit = 4. Upper- and lower-case characters active.
Function codes = 129-143. Control character codes = 1-31.

Appendix

3 1 177	4 P 178	7 J 179	2 4 180	14 5 181	12 6 182	1 7 183	6 8 184	15 9 185	10 0 186	5 - 187
Q 145	W 151	11 E 133	R 146	T 148	Y 153	U 149	I 137	O 143	P 144	/ 187
A 128	S 147	D 132	F 134	G 135	H 136	J 138	K 139	L 140	; 156	13 ENTER
SHIFT	Z 154	10 X 152	C 131	V 150	B 130	N 142	M 141	. 128	. 155	SHIFT
ALPHA LOCK	CTRL	SPACE								FCTN

Figure 3. BASIC Keyboard Scan.

Key-unit = 5. Upper- and lower-case characters active.
Function codes = 1-15. Control character codes = 128-159, 187.

Keyunit = 1					Keyunit = 2					
1 19	2 7	3 8	4 8	5 10	6 19	7 7	8 8	9 8	0 10	-
Q 18	W 4	E 5	R 6	T 11	Y 18	U 4	I 5	O 6	P 11	/ 18
A 1	S 2	D 3	F 12	G 17	H 1	J 2	K 3	L 12	; 17	ENTER
SHIFT	Z 15	x 0	C 14	V 13	B 16	N 15	M 0	. 14	. 13	SHIFT
ALPHA LOCK	CTRL	SPACE								FCTN

Figure 4. Split Keyboard Scan.

Codes returned = 0-19.

CHARACTER CODES FOR SPLIT KEYBOARD

CODES	KEYS*	CODES	KEYS*
0	X,M	10	5,0
1	A,H	11	T,P
2	S,J	12	F,L
3	D,K	13	V. (period)
4	W,U	14	C. (comma)
5	E,I	15	Z,N
6	R,O	16	B. / (slash)
7	2,7	17	G. ; (semicolon)
8	3,8	18	Q,Y
9	4,9	19	1,6

*Note that the first key listed is on the left side of the keyboard,
and the second key listed is on the right side of the keyboard.

Appendix

PATTERN-IDENTIFIER CONVERSION TABLE

Blocks	BINARY CODE (0=off;1=on)	HEXADECIMAL CODE
	0000	0
	0001	1
	0010	2
	0011	3
	0100	4
	0101	5
	0110	6
	0111	7
	1000	8
	1001	9
	1010	A
	1011	B
	1100	C
	1101	D
	1110	E
	1111	F

COLOR CODES

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

Appendix

HIGH-RESOLUTION COLOR COMBINATIONS

The following color combinations produce the sharpest, clearest character resolution on the TI-99/4A color monitor screen. Color codes are included in parentheses.

Black on Medium Green (2, 3)	Light Red on Black (10, 2)
Black on Light Green (2, 4)	Light Red on Dark Red (10, 7)
Black on Light Blue (2, 6)	Dark Yellow on Black (11, 2)
Black on Dark Red (2, 7)	Light Yellow on Black (12, 2)
Black on Cyan (2, 8)	Light Yellow on Dark Red (12, 7)
Black on Medium Red (2, 9)	Dark Green on Light Green (13, 4)
Black on Light Red (2, 10)	Dark Green on Light Yellow (13, 12)
Black on Dark Yellow (2, 11)	Dark Green on Gray (13, 15)
Black on Light Yellow (2, 12)	Dark Green on White (13, 16)
Black on Dark Green (2, 13)	Magenta on Gray (14, 15)
Black on Magenta (2, 14)	Magenta on White (14, 16)
Black on Gray (2, 15)	Gray on Black (15, 2)
Black on White (2, 16)	Gray on Dark Blue (15, 5)
Medium Green on White (3, 16)	Gray on Dark Red (15, 7)
Light Green on Black (4, 2)	Gray on Dark Green (15, 13)
Light Green on White (4, 16)	Gray on White (15, 16)
Dark Blue on Light Blue (5, 6)	White on Black (16, 2)
Dark Blue on Gray (5, 15)	White on Medium Green (16, 3)
Dark Blue on White (5, 16)	White on Light Green (16, 4)
Light Blue on Gray (6, 15)	White on Dark Blue (16, 5)
Light Blue on White (6, 16)	White on Light Blue (16, 6)
Dark Red on Light Yellow (7, 12)	White on Dark Red (16, 7)
Dark Red on White (7, 16)	White on Medium Red (16, 9)
Medium Red on Light Red (9, 10)	White on Light Red (16, 10)
Medium Red on Light Yellow (9, 12)	White on Dark Green (16, 13)
Medium Red on White (9, 16)	White on Magenta (16, 14)
	White on Gray (16, 15)

Appendix

MUSICAL TONE FREQUENCIES

The following table gives frequencies (rounded to integers) of four octaves of the tempered scale (one half-step between notes). While this list does not represent the entire range of tones – or even of musical tones – it can be helpful for musical programming.

<i>Frequency</i>	<i>Note</i>	<i>Frequency</i>	<i>Note</i>
110	A	440	A (above middle C)
117	A [#] ,B ^b	466	A [#] ,B ^b
123	B	494	B
131	C (low C)	523	C (high C)
139	C [#] ,D ^b	554	C [#] ,D ^b
147	D	587	D
156	D [#] ,E ^b	622	D [#] ,E ^b
165	E	659	E
175	F	698	F
185	F [#] ,G ^b	740	F [#] ,G ^b
196	G	784	G
208	G [#] ,A ^b	831	G [#] ,A ^b
220	A (below middle C)	880	A (above high C)
220	A (below middle C)	880	A (above high C)
233	A [#] ,B ^b	932	A [#] ,B ^b
247	B	988	B
262	C (middle C)	1047	C
277	C [#] ,D ^b	1109	C [#] ,D ^b
294	D	1175	D
311	D [#] ,E ^b	1245	D [#] ,E ^b
330	E	1319	E
349	F	1397	F
370	F [#] ,G ^b	1480	F [#] ,G ^b
392	G	1568	G
415	G [#] ,A ^b	1661	G [#] ,A ^b
440	A (above middle C)	1760	A

Error Messages

I. Errors Found When Entering a Line

- * **BAD LINE NUMBER**
 1. Line number or line number referenced equals 0 or is greater than 32767
 2. RESEQUENCE specifications generate a line number greater than 32767
- * **BAD NAME**
 1. The variable name has more than 15 characters
- * **CAN'T CONTINUE**
 1. CONTINUE was entered with no previous breakpoint or program was edited since a breakpoint was taken.
- * **CAN'T DO THAT**
 1. Attempting to use the following program statements as commands: DATA, DEF, FOR, GOTO, GOSUB, IF, INPUT, NEXT, ON, OPTION, RETURN
 2. Attempting to use the following commands as program statements (entered with a line number): BYE, CONTINUE, EDIT, LIST, NEW, NUMBER, OLD, RUN, SAVE
 3. Entering LIST, RUN, or SAVE with no program
- * **INCORRECT STATEMENT**
 1. Two variable names in a row with no valid separator between them (ABC A or A\$A)
 2. A numeric constant immediately follows a variable with no valid separator between them (N 257)
 3. A quoted string has no closing quote mark
 4. Invalid print separator between numbers in the LIST, NUMBER, or RESEQUENCE commands
 5. Invalid characters following CONTINUE, LIST, NUMBER, RESEQUENCE, or RUN commands
 6. Command keyword is not the first word in a line
 7. Colon does not follow the device name in a LIST command
- * **LINE TOO LONG**
 1. The input line is too long for the input buffer

* MEMORY FULL

1. Entering an edit line which exceeds available memory
2. Adding a line to a program causes the program to exceed available memory

II. Errors Found When Symbol Table Is Generated

When RUN is entered but before any program lines are performed, the computer scans the program in order to establish a *symbol table*. A *symbol table* is an area of memory where the variables, arrays, functions, etc., for a program are stored. During this scanning process, the computer recognizes certain errors in the program, as listed below. The number of the line containing the error is printed as part of the message (for example: * BAD VALUE IN 100). Errors in this section are distinguished from those in section III, in that the screen color remains cyan until the symbol table is generated. Since no program lines have been performed at this point, all the values in the *symbol table* will be zero (for numbers) and null (for strings).

* BAD VALUE

1. A dimension for an array is greater than 32767
2. A dimension for an array is zero when OPTION BASE = 1

* CAN'T DO THAT

1. More than one OPTION BASE statement in your program
2. The OPTION BASE statement has a higher line number than an array definition

* FOR-NEXT ERROR

1. Mismatched number of FOR and NEXT statements

* INCORRECT STATEMENT

DEF

1. No closing ")" after a parameter in a DEF statement
2. Equals sign (=) missing in DEF statement
3. Parameter in DEF statement is not a valid variable name

Error Messages

DIM

4. DIM statement has no dimensions or more than three dimensions
5. A dimension in a DIM statement is not a number
6. A dimension in a DIM statement is not followed by a comma or a closing ")"
7. The *array-name* in a DIM statement is not a valid variable name
8. The closing ")" is missing for array subscripts

OPTION BASE

9. OPTION not followed by BASE
10. OPTION BASE not followed by 0 or 1

* MEMORY FULL

1. Array size too large
2. Not enough memory to allocate a variable or function

* NAME CONFLICT

1. Assigning the same name to more than one array (DIM A(5), A(2,7))
2. Assigning the same name to an array and a simple variable
3. Assigning the same name to a variable and a function
4. References to an array have a different number of dimensions for the array (B=A(2,7)+2, PRINT A(5))

III. Errors Found When a Program Is Running

When a program is running, the computer may encounter statements that it cannot perform. An error message will be printed, and unless the error is only a warning the program will end. At that point, all variables in the program will have the values assigned when the error occurred. The number of the line containing the error will be printed as part of the message (for example: CAN'T DO THAT IN 210).

* BAD ARGUMENT

1. A built-in function has a bad argument
2. The string expression for the built-in functions ASC or VAL has a zero length (null string)
3. In the VAL function, the string expression is not a valid representation of a numeric constant

* BAD LINE NUMBER

1. Specified line number does not exist in ON, GOTO or GOSUB statement
2. Specified line number in BREAK or UNBREAK does not exist (warning only)

* BAD NAME

1. Subprogram name in a CALL statement is invalid

* BAD SUBSCRIPT

1. Subscript is not an integer
2. Subscript has a value greater than the specified or allowed dimensions of an array
3. Subscript 0 used when OPTION BASE 1 specified

* BAD VALUE

CHAR

1. *Character-code* out of range in CHAR statement
2. Invalid character in *pattern-identifier* in CHAR statement

CHR\$

3. Argument negative or larger than 32767 in CHR\$

COLOR

4. *Character-set-number* out of range in COLOR statement
5. *Foreground* or *background color code* out of range in COLOR statement

EXPONENTIATION (^)

6. Attempting to raise a negative number to a fractional power

FOR

7. Step increment is zero in FOR-TO-STEP statement

HCHAR, VCHAR, GCHAR

8. *Row* or *column-number* out of range in HCHAR, VCHAR, or GCHAR statement

JOYST, KEY

9. *Key-unit* out of range in JOYST or KEY statement

ON

10. *Numeric-expression* indexing *line-number* is out of range

Error Messages

OPEN, CLOSE, INPUT, PRINT, RESTORE

11. *File-number* negative or greater than 255
12. Number-of-records in the SEQUENTIAL option of the OPEN statement is non-numeric or greater than 32767
13. *Record-length* in the FIXED option of the OPEN statement is greater than 32767

POS

14. The *numeric-expression* in the POS statement is negative, zero, or larger than 32767

SCREEN

15. Screen *color-code* out of range

SEG\$

16. The value of *numeric-expression1* (character position) or *numeric-expression2* (length of substring) is negative or larger than 32767

SOUND

17. *Duration, frequency, volume or noise* specification out of range

TAB

18. The value of the character position is greater than 32767 in the TAB function specification

* CAN'T DO THAT

1. RETURN with no previous GOSUB statement
2. NEXT with no previous matching FOR statement
3. The *control-variable* in the NEXT statement does not match the *control-variable* in the previous FOR statement
4. BREAK command with no line number

* DATA ERROR

1. No comma between items in DATA statement
2. *Variable-list* in READ statement not filled but no more DATA statements are available
3. READ statement with no DATA statement remaining

4. Assigning a string value to a numeric variable in a READ statement
5. *Line-number* in RESTORE statement is greater than the highest line number in the program

* FILE ERROR

1. Attempting to CLOSE, INPUT, PRINT, or RESTORE a file not currently open
2. Attempting to INPUT records from a file opened as OUTPUT or APPEND
3. Attempting to PRINT records on a file opened as INPUT
4. Attempting to OPEN a file which is already open

* INCORRECT STATEMENT

General

1. Opening "(" , closing ")" , or both missing
2. Comma missing
3. No line number where expected in a BREAK, UNBREAK, or RESTORE (BREAK 100.)
4. "+" or "-" not followed by a numeric expression
5. Expressions used with arithmetic operators are not numeric
6. Expressions used with relational operators are not the same type
7. Attempting to use a string expression as a subscript
8. Attempting to assign a value to a function
9. Reserved word out of order
10. Unexpected arithmetic or relational operator is present
11. Expected arithmetic or relational operator missing

Built-in Subprograms

12. In JOYST, the *x-return* and *y-return* are not numeric variables
13. In KEY, the *key-status* is not a numeric variable
14. In GCHAR, the third specification must be a numeric variable
15. More than three tone specifications or more than one noise specification in SOUND
16. CALL is not followed by a subprogram name

Error Messages

File Processing-Input/Output Statements

17. Number sign (#) or colon (:) in *file-number* specification for OPEN, CLOSE, INPUT, PRINT, or RESTORE is missing
18. *File-name* in OPEN or DELETE must be a string expression
19. A keyword in the OPEN statement is invalid or appears more than once
20. The number of records in SEQUENTIAL option is less than zero in the OPEN statement
21. The record length in the FIXED option in the OPEN statement is less than zero or greater than 255
22. A colon (:) in the CLOSE statement is not followed by the keyword DELETE
23. *Print-separator* (comma, colon, semicolon) missing in the PRINT statement where required
24. *Input-prompt* is not a string expression in INPUT statement
25. *File-name* is not a valid string expression in SAVE or OLD command

General Program Statements

FOR

26. The keyword FOR is not followed by a numeric variable
27. In the FOR statement, the *control-variable* is not followed by an equals sign (=)
28. The keyword TO is missing in the FOR statement
29. In the FOR statement, the *limit* is not followed by the end of line or the keyword STEP

IF

30. The keyword THEN is missing or not followed by a line number

LET

31. Equals sign (=) missing in LET statement

NEXT

32. The keyword NEXT is not followed by *control-variable*

ON-GOTO, ON-GOSUB

33. ON is not followed by a valid numeric expression

RETURN

34. Unexpected word or character following the word RETURN

User-Defined Functions

35. The number of function arguments does not match the number of parameters for a user-defined function

* INPUT ERROR

1. Input data is too long for Input/Output buffer (if data entered from keyboard, this is only a warning – data can be re-entered)
2. Number of variables in the *variable-list* does not match number of data items input from keyboard or data file (warning only if from keyboard)
3. Non-numeric data INPUT for a numeric variable. This condition could be caused by reading padding characters on a file record. (Warning only if from keyboard)
4. Numeric INPUT data produces an overflow (warning only if from keyboard)

- * I/O ERROR – This condition generates an accompanying error code as follows:

When an I/O error occurs, a two-digit error code (XY) is displayed with the message:

* I/O ERROR XY IN *line-number*

The first digit (X) indicates which I/O operation caused the error.

<i>X Value</i>	<i>Operation</i>
0	OPEN
1	CLOSE
2	INPUT
3	PRINT
4	RESTORE
5	OLD
6	SAVE
7	DELETE

Error Messages

The second digit (Y) indicates what kind of error occurred.

<i>Y Value</i>	<i>Error Type</i>
0	Device name not found (Invalid device or file name in DELETE, LIST, OLD, or SAVE command)
1	Device write protected (Attempting to write to a protected file)
2	Bad open attribute (One or more OPEN options are illegal or do not match the file characteristics)
3	Illegal operation (Input/output command not valid)
4	Out of space (Attempting to write when insufficient space remains on the storage medium)
5	End of file (Attempting to read past the end of a file)
6	Device error (Device not connected, or is damaged. This error can occur during file processing if an accessory device is accidentally disconnected while the program is running.)
7	File error (The indicated file does not exist or the file type — program file or data file — does not match the access mode.)

• MEMORY FULL

1. Not enough memory to allocate the specified character in CHAR statement
2. GOSUB statement branches to its own *line-number*
3. Program contains too many pending subroutine branches with no RETURN performed
4. Program contains too many user-defined functions which refer to other user-defined functions
5. Relational, string, or numeric expression too long
6. User-defined function references itself

• NUMBER TOO BIG (warning given — value replaced by computer limit as shown below)

1. A numeric operation produces an overflow (value greater than 9.999999999999999E127 or less than -9.999999999999999E127)
2. READING from DATA statement results in an overflow assignment to a numeric variable
3. INPUT results in an overflow assignment to a numeric variable

• STRING-NUMBER MISMATCH

1. A non-numeric argument specified for a built-in function, tab-function, or exponentiation operation
2. A non-numeric value found in a specification requiring a numeric value
3. A non-string value found in a specification requiring a string value
4. Function argument and parameter disagree in type, or function type and expression type disagree for a user-defined function
5. *File-number* not numeric in OPEN, CLOSE, INPUT, PRINT, RESTORE
6. Attempting to assign a string to a numeric variable
7. Attempting to assign a number to a string variable

Note: Additional error codes may occur when you are using various accessories, such as the TI Disk Memory System or Solid State Thermal Printer, with the computer. Consult the appropriate device owner's manual for more information on these error codes.

IV. Error Returned When an OLD Command Is Not Successful

*CHECK PROGRAM IN MEMORY

The OLD command does not clear program memory unless the loading operation is successful. If an OLD command fails or is interrupted, however, any program currently in memory may be partially or completely overwritten by the program being loaded. LIST the program in memory before proceeding.

Glossary

Words in *italics* and followed by an “*” are also referenced in the Glossary. Note that words with an [*] after them are defined in the *Texas Instruments TI-99/4A User's Reference Guide* and are reproduced here with permission of Texas Instruments, Inc.

ABS—the absolute value function of BASIC. The ABS function returns the magnitude of its *arguments**. See *magnitude**. For example:

```
PRINT ABS (-2.5)
2.5
```

returns a value of 2.5 as the magnitude of 2.5.

absolute value function—see *ABS**.

accessory devices[*]—additional equipment which attaches to the computer and extends its functions and capabilities. Included are preprogrammed *Command Modules** and units which send, receive or store computer data, such as printers and disks. These are often called peripherals.

address—the identification of a memory location in the computer. For example, your computer has 16K addresses which identify each of the 16K bytes of *random access memory** in the TI-99/4A. See also *byte**.

algorithm—a way to solve a problem in a finite number of steps. For example, a computer program is an algorithm. A recipe is also an algorithm.

alphanumeric—a contraction of the term “alphabetic and numeric”. The alphanumeric keys are those which produce either a letter of the alphabet, A-Z, or a numeral: 0,1,2,3,4,5,6,7,8,9.

arctangent function—see *ATN**.

argument—the value you supply to a function which enables it to produce an output. The arguments of functions are usually enclosed in parentheses following the function. For example:

```
PRINT TAB(3)
```

arithmetic expression—a group of arithmetic operators and operands. For example, the following are some arithmetic expressions, where A and B are variables

```
2 + 2
3 × 6-8
A-3 × B
8.3 + (B × 7-3 / 2) / A
```

arithmetic operator—a symbol used for arithmetic operations. For example

^	exponentiation
-	negation, also called unary minus
×	multiplication
/	division
+	addition
-	subtraction, also called binary minus

Note that the same symbol is used for negation as for subtraction.

Besides the arithmetic operators, there are also *logical operators**, and *concatenation**.

array[*]—a collection of numeric or string variables, arranged in a list or matrix for processing by the computer. Each element in an array is referenced by a *subscript** describing its position in the list.

ASC—a BASIC string function which produces the ASCII code when a character or string expression which reduces to a character is given as its argument. ASC can be used as either a direct command or in a program statement. As an example of its use,

```
PRINT ASC("A")
65
```

returns a value of 65 as the ASCII code for "A". ASC and CHR\$* are *inverse functions**.

ASCII[*]—the American Standard Code for Information Interchange, the code structure used internally in most personal computers to represent letters, numbers, and special characters.

assembly language—a higher level language than machine language. A program called an assembler converts assembly language instructions into machine language. Assembly language is much more convenient for people to use than trying to program in machine language directly.

ATN—the ATN function returns the angle in radians when you supply the tangent of that angle as the argument of ATN. For example, if the tangent is .5463024898, then

```
PRINT ATN(.5463024898)
.5
```

shows that the angle .5 radians has a tangent of .5463024898. The tangent function and arctangent function are inverse functions, that is, for any angle X, the following identity holds

$$X = \text{ATN}(\text{TAN}(X))$$

auto repeat—a contraction of the term "automatic repeat". If you hold down any key which produces a character, the computer will automatically keep generating that character. The auto repeat feature acts as if you pressed the same key many times.

base—the number which is raised to a power. For example, the EXP* function uses the number e. Another definition of base refers to number systems. For example, ten is the base of our ordinary decimal system, while two is the base of the binary system. The reason these numbers are called the base of a number system is that any number can be expressed as the sum of powers of the base. For example:

$$521 = 5 \times 10^2 + 2 \times 10^1 + 1 \times 10^0$$

background color—the background color surrounding a character. See also *foreground color** and CALL COLOR*.

BASIC[*]—an easy-to-use popular programming language used in most personal computers. The word BASIC is an acronym for "Beginners All-purpose Symbolic Instruction Code."

baud[*]—commonly used to refer to bits per second.

binary[*]—a number system based on two digits, 0 and 1. The internal language and operations of the computer are based on the binary system.

binary number—see *binary**.

bit—contraction of the term binary digit. In the binary number system, only the numerals 0 and 1 are allowed as digits. See also *binary**.

branch[*]—a departure from the sequential performance of program statements. An unconditional branch causes the computer to jump to a specific program line every time the branching statement is encountered. A conditional branch transfers program control based on the result of some arithmetic or logical operation.

BREAK—a BASIC word which can be used in either a command or statement to interrupt the execution of a program. The arguments of BREAK specify the line numbers before which execution will stop. You can use the CON* (see CONTINUE*) command to resume execution after a *breakpoint** if you have not edited the program.

breakpoint[*]—a point in the program specified by the BREAK* command where program execution can be suspended. During a breakpoint, you can perform operations in the *Command Mode** to help you locate program errors. Program execution can be resumed with a CONTINUE* command, unless editing took place while the program was stopped.

BREAKPOINT—when the execution of a program is interrupted by either (1) the FCTN and CLEAR keys being pressed, or (2) execution of a BREAK statement, or (3) execution of a BREAK command, the message

```
* BREAKPOINT AT LINE XX
```

is printed, where XX is the line number where execution is interrupted. See also BREAK*.

buffer[*]—an area of computer memory for temporary storage of an input or output record.

bug[*]—a hardware defect or programming error which causes the intended operation to be performed incorrectly.

BYE—a BASIC word which exits you from BASIC. Useful with a disk system.

byte[*]—a string of *binary** digits (bits) treated as a unit, often representing one date *character**. The computer's memory capacity is often expressed as the number of bytes available. For example, a computer with 16K bytes of memory has about 16,000 bytes available for storing programs and data.

CALL CHAR—a BASIC subprogram which allows you to define graphics characters. The general form is

CALL CHAR (character-code,"pattern identifier")

The character-code is specified by an ASCII code in the range 32-159. The pattern identifier is specified using hexadecimal codes for the left and right blocks of the character. See also the Appendix of the Pattern-Identifier Conversion Table.

CALL CLEAR—a BASIC subprogram which clears the screen.

CALL COLOR—a BASIC subprogram which allows you to specify the foreground and background color for groups of characters. The general form of the subprogram is

`CALL COLOR (character-set-number,foreground-color-code,background-color-code)`

The character-set-codes for different groups are shown in the Appendix of ASCII Character Codes. The color codes for foreground and background colors are given in the Appendix of Color Codes.

CALL GCHAR—a BASIC subprogram that returns the character located at a given position on the screen. The general form is

`CALL GCHAR(row-number,column-number,numeric-variable)`

You specify the row and column number, and CALL GCHAR returns the ASCII character code of the character at that position in the numeric variable. The row and column numbers can also be numeric expressions. Any legal variable can be used for the numeric variable.

CALL HCHAR—a BASIC subprogram which prints a character at a position on the screen specified by the row and column numbers. The general form is

`CALL HCHAR(row-number,column-number,character-code,number-of-repetitions)`

The default value for the argument number-of-repetitions is 1, so it's not necessary to specify this argument if you only want to print one character. The CALL HCHAR prints repetition characters horizontally. For example, the command

`CALL HCHAR(12,16,30,10)`

will print 10 cursor blocks across the screen starting at row 12, column 16. Numeric expressions may be used for the arguments.

The screen is considered to have rows labeled 1-24, and columns 1-32. The valid character-codes and repetitions are 0-32767.

CALL SCREEN—a BASIC subprogram that changes the screen color according to a numeric code specified by the argument. For example

`CALL SCREEN(7)`

changes the screen background color to dark red. If you execute this subprogram as a direct command, the border around your screen will change color for just a second. If CALL SCREEN is executed as a statement, as in

`10 CALL SCREEN(7)`

and RUN, you will see the entire screen change color. A complete list of the color codes for the argument of CALL SCREEN is given in the Appendix of Color Codes.

CALL SOUND—a subprogram which generates a sound. For example,

`CALL SOUND(1000,300,1)`

produces a tone which lasts 1000 msec (1 second), of frequency 300 Hz (see *Hertz**) and volume 1. A tone is a sound of a single frequency. The general form of CALL SOUND is

`CALL SOUND(duration,frequency 1,volume 1,frequency 2,volume 2,frequency 3,volume 3,frequency 4,volume 4)`

where the frequencies and volumes for 2, 3, and 4 are optional arguments. A negative frequency value in the range -1 to -8 produces a predefined noise.

The duration of the sound may be specified from 1 to 4250 msec, the frequency from 110 to 44733 Hz, and the volume from 0 (loudest) to 30 (quietest). A maximum of three tones and one noise can be generated simultaneously.

CALL VCHAR—a BASIC subprogram which prints a character at a position on the screen specified by the row and column numbers. The general form is

`CALL VCHAR (row-number,column-number,character-code,number-of-repetitions)`

The default value for the argument number-of-repetitions is 1, so it's not necessary to specify this argument if you only want to print one character. The CALL VCHAR prints repetition characters vertically. For example, the direct command

`CALL VCHAR (12,16,30,10)`

will print 10 cursor blocks down the middle of the screen starting at row 12, column 16.

The screen is considered to have rows labeled 1-24, and columns 1-32. The valid character-codes and repetitions are 0-32767.

CALL JOYST—a BASIC subprogram which allows you to input data via the optional Wired Remote Controllers.

CALL KEY—a BASIC subprogram which allows the computer to read the keyboard and determine what key is depressed or has been released. The general form is

`CALL KEY(key-unit,return-variable,status-variable)`

The key-unit values may be 0-5. For more information on how these values set up the keyboard, see the Appendix on Keyboard Mapping. The return-variable is given a numeric character code corresponding to the key pressed. The status-variable returns information concerning what has happened since the last CALL KEY execution.

character function—see *CHR\$**.

character[*]—a letter, number, punctuation symbol, or special graphics symbol.

CHR\$—a BASIC string function which can be used as either a direct command or in a statement. CHR\$ produces a character when a number or numeric expression is given as its argument. For example

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

prints the letter "A" on your screen. The CHR\$ and ASC* functions are *inverse functions**. That is, for any number X,

```
X=CHR$(ASC(X))
```

See also the Appendix of ASCII Character Codes.

CLEAR—pressing the FCTN and "4" key activates the CLEAR feature. The "4" key is referred to as the CLEAR key because the printed scale that slides over the top row of keys is labeled "CLEAR" over the "4" key. When a program is running, the FCTN and CLEAR keys will interrupt execution. If you are typing in program lines using Number Mode (see *NUM**) the FCTN and CLEAR will exit you from Number Mode.

CLOSE—a BASIC word used in a direct command or statement to close files that have been opened (see *OPEN**). The argument of CLOSE is the file number to be closed. For example

```
CLOSE #1
```

will close file number 1. Note that the "#" sign must precede the file number.

command[*]—an instruction which the computer performs immediately. Commands are not a part of a program and thus are entered with no preceding line number.

Command Mode[*]—when no program is running, the computer is in the Command (or Immediate) Mode and performs each task as it is entered.

command modules[*]—preprogrammed *ROM** modules which are easily inserted in the TI computer to extend its capabilities.

common logs—logarithms to base 10 (see *base**, and *LOG**). To convert from natural logs to common logs, just divide by LOG(10). For example, to print the common log of 2, do

```
PRINT LOG(2)/LOG(10)
.3010299957
```

compiler—a program that translates or compiles *source** code written in a high-level language into machine language. While BASIC translates one line at a time, a compiler translates the entire program into machine language. This machine language version of the high-level program can then be executed. The code produced by a compiler executes much faster than a BASIC interpreter because the BASIC interpreter must translate each program line into machine code every time the line is encountered. In contrast, the compiler does the translation only once. Compilers are commonly used for languages such as FORTRAN and COBOL.

compound interest—rather than paying interest only once a year, banks may pay you interest by the month or even daily. Your money grows faster this way. Of course, you also pay more interest to a bank under compound interest if you borrow money.

computer animation—producing the illusion of movement (animation) by rapidly showing a sequence of computer generated images.

CON—see *CONTINUE**.

concatenation[*]—linking two or more *strings** to make a longer string. The "&" is the concatenation operator.

conditional branch—also called a conditional jump. Test for the truth or falseness of a relation before branching to a line. An IF-THEN-ELSE statement is an example of a conditional branch. See also *branch**, *IF**, and *ON**.

constant[*]—a specific numeric or *string** value. A numeric constant is any real number, such as 1.2 or -9054. A string constant is any combination of up to 112 characters enclosed in quotes, such as "HELLO THERE" or "275 FIRST ST."

CONTINUE—long form of CON (see *CON**). The CON or CONTINUE command resumes execution after a breakpoint if you have not changed the program. See also *BREAK** and *BREAKPOINT**.

control characters—a term for *control codes**. Although there are no pre-defined ASCII printable characters for codes 0-29, they can be accessed through the *CHR\$** function just as if they were printable.

control codes—codes which are not assigned to a printable character. Control codes are normally used in data communications, and to control devices such as printers. The standard control codes for the TI-99/4A have code numbers 0-29. See also the Appendices on Function and Control Key Codes, and Keyboard Mapping for typical applications and information on how to generate these codes from the keyboard.

control variable—see *FOR**.

COS—the COS function returns the cosine of its argument in radians. The cosine of a right triangle is defined as the side adjacent the angle to the hypotenuse. For example:

```
PRINT COS(.5)
.8775825619
```

cosine function—see *COS**.

cursor[*]—a symbol which indicates where the next *character** will appear on the screen when you press a key.

data[*]—basic elements of information which are processed or produced by the computer.

DATA—a BASIC word used in statements which contain the data read by *READ* statements**

database—a collection of items. Sometimes spelled "data base".

datafiles—files of data. Sometimes spelled "data files".

debugging—to remove the bugs in a program or equipment. The term bugs refers to any problems or errors which prevents your program or equipment from working properly.

decrement—to decrease. Commonly used to mean decrease by one.

DEF—a BASIC word used in a statement that allows you to define your own function. For example,

```
10 DEF SQUARE(X)=X*X
20 PRINT SQUARE(2)
```

Line 10 defines the function "SQUARE". Line 20 prints the value of SQUARE when its argument is 2.

default[*]—a standard characteristic or value which the computer assumes if certain specifications are omitted within a *statement** or a *program**.

DELETE—a BASIC word used to delete files from a storage device such as a disk.

delimit—to specify the limits.

destructive cursor—a cursor which erases every character it passes over.

device[*]—(see *accessory devices*).

DIM—a BASIC word used to declare dimensioned variables. The argument of DIM is the array name and its maximum index. For example,

```
DIM N(100)
```

declares an array called N consisting of the *elements* N(0), N(1), N(2), . . . , N(100)*. The three dots indicate that the elements N(3) to N(99) are also declared. The computer allocates space for each element of a numeric array. However, each element of a string array is set to no characters (the null string). Up to three dimensions or indexes can be defined for an array. The default value for each index is 10. So it's not really necessary to do a

```
DIM N(10)
```

or

```
DIM N(10,10)
```

or

```
DIM N(10,10,10)
```

More than one array name can be specified by a DIM statement.

dimensioned variables—variables which are accessed by a number or numeric expression as well as a name. Dimensioned variables must be declared by a DIM statement (see *DIM**).

direct command—an instruction to the computer which is immediately executed. Sometimes called just "command". That is, a direct command has no line number as does a statement. In some commands, such as TRACE and BREAK the action of the command is not apparent to you until the program starts to execute.

disk[*]—a mass storage device capable of random and sequential access.

DISPLAY—a BASIC word used in statements or direct commands to print information on the screen. Equivalent to PRINT when printing to the screen.

display[*]—(noun) the video screen; (verb) to cause characters to appear on the screen.

display memory—a special portion of the computer memory which contains information about the characters on the screen. The display memory contains the ASCII code, foreground color, and background color of each character.

dummy argument—an argument whose name does not matter.

duration—the time that a sound produced by *CALL SOUND** lasts.

dynamic—changing

editing—verb form of “edit,” which means to alter or change. Editing a program may involve the addition, replacement or deletion of characters.

Edit Mode[*]—the mode used to change existing program lines. The EDIT mode is entered by using the Edit Command or by entering the line number followed by SHIFT ↓ or SHIFT ↑. The line specified is displayed on the screen and changes can be made to any *character** using the editing keys.

element—an item of an *array**.

END—a BASIC word which causes the computer to stop execution. The END is commonly the highest numbered statement in a program. Use of END is optional since the computer automatically stops after executing the line with the highest line number. See also *STOP**.

end-of-file[*]—the condition indicating that all *data** has been read from a *file**.

ENTER—a key which, when pressed, tells the computer that you’re done entering a line of input

EOF—a BASIC function which determines if the end-of-file has been reached. Not used with cassette tape.

execute[*]—to run a program; to perform the task specified by a *statement** or *command**.

EXP—the exponential function of BASIC. This function raises the number e to the power given by its argument. The argument is the exponent of e. For example:

```
PRINT EXP(1)
2.718281828
```

see also *LOG**.

exponent[*]—a number indicating the power to which a number or *expression** is to be raised; usually written at the right and above the number. For example, $2^8 = 2 \times 2$. In TI BASIC the exponent is entered following the ^ symbol or following the letter “E” in *scientific notation**. For example, $2^8 = 2 \wedge 8$; $1.3 \times 10^{25} = 1.3E25$.

exponential—another term for *scientific notation*. See also *exponent*.

exponential function—see *EXP**.

exponentiation—the process of raising a number to a power. See also *exponent*.

expression[*]—a combination of constants, variables, and operators which can be evaluated to a single result. Included are numeric, string, and relational expressions.

FCTN—a key which is used with other keys to provide extra functions. For example, pressing the FCTN key and then a key with a printable symbol on its side will print that character. Thus FCTN and the “I” key will print a question mark “?”.

field—a group of 14 columns

file[*]—a collection of related data records stored on a device; also used interchangeably with *device** for input/output equipment which cannot use multiple files, such as a line printer.

FIXED—a BASIC word used in an OPEN statement to specify that the records are all the same length. The optional argument of FIXED may be a number or numeric expression which tells the maximum length of a record.

fixed-length records[*]—records in a *file** which are all the same length. If a file has fixed-length records of 95 characters, each record will be allocated 95 *bytes** even if the *data** occupies only 76 positions. The computer will add padding characters on the right to ensure that the record has the specified length.

flowchart—a diagram consisting of graphical symbols and arrows which describe the logical decisions and flow of data in a program.

FOR—part of a FOR-NEXT loop. The FOR-NEXT statements cause controlled looping between statements. For example,

```
10 FOR I=1 TO 10 STEP 2
20 PRINT I
30 NEXT I
40 PRINT "DONE"
```

causes the numbers 1, 3, 5, 7, and 9 to be printed on your screen. The control variable I is initially set to the lower limit of 1. Line 20 prints I and line 30 adds the STEP size of 2 to I and then checks to see if this new value of I exceeds the upper limit of 10. If not, the computer starts execution again from the line following the FOR. If the new value of I exceeds the upper limit, then the computer starts executing the statement after the NEXT. Any legal variable name can be used for the control variable.

foreground color—the color of a character. See also *background color**, and *CALL COLOR**.

format—the manner or appearance in which something, such as output, is presented.

frequency—1 divided by the period of a wave. The frequency specifies the number of occurrences of a wave’s peak amplitude per second for a wave of a single frequency.

function[*]—a feature which allows you to specify as "single" operations a variety of procedures, each of which actually contains a number of steps; for example, a procedure to produce the square root via a simple reference name.

GOSUB—a BASIC word used in a statement which directs the computer to execute the subroutine specified by the argument of GOSUB. For example, the statement

```
20 GOSUB 100
```

directs the computer to execute the subroutine starting at line 100. See also *RETURN**.

GOTO—a BASIC word used in statements to make the computer GOTO a line. The argument of GOTO is the next line that the computer will execute after the GOTO. For example:

```
10 GOTO 30
20 PRINT
30 PRINT 2+2
```

the "GOTO 30" of line 10 will make the computer skip line 20 and execute line 30 next.

graphics[*]—visual constructions on the screen, such as graphs, patterns, and drawings, both stationary and animated. TI BASIC has built-in subprograms which provide easy-to-use color graphic capabilities.

graphics line[*]—a 32-character line used by the TI BASIC graphics subprograms.

grid—a rectangular arrangement of lines. For example, the pattern formed by the wires of a screen door is an example of a grid. The horizontal and vertical coordinates of points you can print at on the TV screen using *CALL HCHAR** and *CALL VCHAR** define a grid.

hardware[*]—the various devices which comprise a computer system, including memory, the keyboard, the screen, disk drives, line printers, etc.

Hertz—a unit of frequency named after the famous 19th century physicist Heinrich Hertz. A Hertz equals one cycle per second and is abbreviated Hz.

hexadecimal[*]—a base-16 number system using 16 symbols, 0-9 and A-F. It is used as a convenient "shorthand" way to express *binary** code. For example, 1010 in binary = A in hexadecimal, 11111111 = FF. Hexadecimal is used in constructing patterns for graphics characters in the *CALL CHAR* subprogram.

Hz—see *Hertz**.

IF—a BASIC word used in a statement with *THEN* and optionally *ELSE* to specify a *conditional branch** in a program. The general form is

```
IF relation THEN line number XX ELSE line number YY
```

The IF statement says that if the relation is true, the computer should execute line number XX next. If the relation is false, the computer executes line number YY next. For example,

```
20 IF A=B THEN 100 ELSE 60
```

Line 20 says that if the variable A equals the variable B, the computer should execute line 100 next. If A is unequal to B, the computer will execute line 60 next. Without the *ELSE*, the computer would just execute the line following 20 if the relation was false.

immediate mode[*]—see *Command Mode**.

increment[*]—a positive or negative value which consistently modifies a *variable**.

incremented—see *increment**.

index number—a number or numeric expression which specifies a specific variable in an *array**. Another name for an index number is *subscript**.

index variable—a *variable** which is used to index or select another quantity. For example, in the dimensioned variable, A(I), the variable I is called the index variable because it selects a specific A(I). See also *dimensioned variables**.

Individual Retirement Account (IRA)—a method by which you save money toward your retirement and not pay taxes until you start collecting. For example, an employed person can contribute up to \$2000 a year toward an IRA. You do not pay taxes on the money contributed or the interest until the money is withdrawn which you can start at age 59½. For more details, check with any bank.

infinite loop—a loop that will never terminate unless the program is interrupted. For example, the program consisting of the single line

```
10 GOTO 10
```

forms an infinite loop.

initial value—the first value that a variable is given.

inner loop—a loop which lies entirely inside of another.

INPUT—a BASIC word used in a statement which halts execution to allow you to enter input from the keyboard. For example, the program

```
10 INPUT B
20 PRINT 2*B
```

will halt when line 10 is executed. You'll see a "?" to prompt you to enter a value for B. Optionally, you can include some text to be printed by enclosing it within quotes. For example,

```
10 INPUT"NUMBER=?":B
20 PRINT 2*B
```

The general form is

```
INPUT "Text":List of Variables
```

Actually, the text string can be a string variable. For example,

```
10 A$="NUMBERS=?"
20 INPUT A$:A,B
30 PRINT A+B
```

will also work. The INPUT is also used to input data from a storage device.

input[*]—(noun) *data** to be placed in computer memory; (verb) the process of transferring data into memory.

input line[*]—the amount of *data** which can be entered at one time. In TI BASIC, this is 112 characters.

INT—the integer function of BASIC. This function truncates or cuts off the decimal portion of the argument of INT. For example;

```
PRINT INT(2.5)
2
```

returns a 2 because the decimal part, .5, is removed by INT.

integer function—see *INT*.

integer[*]—a whole number, either positive, negative, or zero.

INTERNAL—a BASIC word used in an *OPEN** statement to tell the computer to record data in internal machine format. This format is more efficient for the computer.

internal data-format[*]—*data** in the form used directly by the computer. Internal numeric data is 8 bytes* long plus 1 byte which specifies the length. The length for internal string data is one byte per character in the *string** plus one length-byte.

interpreter—a program which interprets or converts each line of the source code to machine code as it is encountered. The BASIC in your computer is an interpreter.

inverse functions—functions which are the opposite of one another. For example

```
X=EXP(LOG(X))
```

for any number X, the above is an identity because that EXP and LOG functions are inverse functions. Likewise, the CHR\$ and ASC functions are inverse functions since for any character X\$ within the range of their allowed arguments

```
X$=CHR$(ASC(X$))
```

For example, enter the following direct command:

```
PRINT CHR$(ASC("A"))
```

and you'll see an "A" printed. See also the Appendix of ASCII Character Codes.

I/O[*]—Input/Output; usually refers to a device function. I/O is used for communication between the computer and other devices (e.g., keyboard, disk).

iteration[*]—the technique of repeating a group of program statements; one repetition of such a group. See *loop**.

key-unit—the first argument of *CALL KEY**. The key-unit defines the mode in which the keyboard is scanned. See the Appendix of Keyboard Mapping.

kilobyte—1024 bytes. Commonly abbreviated as K byte. For example, your TI-99/4A computer has 16 K bytes of *RAM**.

LEN—a BASIC string function which returns the length of a string used as its argument. For example,

```
PRINT LEN("AA")
2
```

returns a value of 2 because there are two characters in the string "AA".

length function—see *LEN**.

LET—a BASIC word which assigns a value to a variable. LET is optional and so is usually not explicitly typed in. For example,

```
LET X=1
```

and

```
X=1
```

both assign the value 1 to X. You can assign values to variables by either a direct command or in a statement.

level—the level of detail in a pseudocode program. Higher levels have less detail than low level descriptions. The term level is also applied to the description of computer languages. A high level lan-

guage such as BASIC is much closer to English than assembly language or machine language. See also *BASIC**, *assembly language**, and *machine language**.

limit—the end or border of a quantity.

line[*]—see *graphics line*, *input line*, *print line*, or *program line*.

linear array—dimensioned variables with a single index. For example, N(I) is a linear array. The term "linear array" comes from a geometric analogy to a straight line of array elements since only one index is used per element.

LIST—a BASIC word which lists the program in memory. The general form of LIST is
LIST starting line-final line

If no arguments are given for the starting and final line, the LIST command lists the entire program. If the starting line is absent, the program lists up to the final line. For example,

```
LIST -300
```

will list from the lowest line number up to and including line 300. If the final line is not specified, the program lists until the end of the program. For example,

```
LIST 300-
```

will list from line 300 to the end.

The LIST command can also be used to list a program to a device through the RS-232-C interface. For example, if you have a printer operating at 1200 baud connected via the RS-232 C interface, the command

```
LIST "RS232.BA=1200"
```

will print your program on the printer.

literal—a character or symbol that has no special meaning to the computer. For example, "A", "B", "!", "#", etc. have no special meaning nor does the string of literals "MONEY". However, "PRINT", "LIST", "*", "/", do have special meaning to the computer when used as a command or statement. These words or symbols with special meaning to the computer are called tokens. Putting these tokens within quotes turns off their special meaning. For example

```
PRINT "PRINT"
```

uses the command PRINT to print the string of literals "PRINT."

load—the process of loading a program from a storage device such as cassette tape, wafer tape or disk back into the memory of the computer. For a cassette tape, type in

```
OLD CS1
```

and press the ENTER key.

LOG—the BASIC function which returns the natural logarithm (log) of its argument. The natural log is to the base e. In general, for the number y which equals e raised to the X'th power

$$Y = e^X$$

then X is the log of Y. In other words $X = \text{LOG}(Y)$. For example:

```
PRINT LOG (1)
```

```
0
```

The log of 1 is 0 because

$$1 = e^0$$

where $Y = 1$ and $X = 0$.

The EXP and LOG functions are inverse functions of each other. That is, for any number Y,

$$Y = \text{LOG} (\text{EXP}(Y))$$

For example:

```
PRINT LOG (EXP(1))
```

```
1
```

since

$$Y = e^{\text{LOG} (Y)}$$

is an identity. That is, both sides of this equation always match for any number Y.

logical operators—test for the truth or falseness of relations. The logical operators are:

```
= equal
< less than
> greater than
<> unequal
<= less than or equal to
>= greater than or equal to
```

The result of a logical test returns the value "-1" if the result is true or "0" if the result is false. For example,

```
PRINT 3>2
```

```
-1
```

```
PRINT "AB">="A"
```

```
-1
```

are both true, while
 PRINT 3=2
 0

is false.

loop[*]—a group of consecutive program lines which are repeatedly performed, usually a specified number of times.

machine language—the lowest level language that the computer understands. Machine language consists of bits (see *bits**) that tell the computer exactly what to do. A high-level language like BASIC must interpret each BASIC command or statement into a low-level code that the computer understands. See also *BASIC** and *level**. Machine language is very hard for most people to directly write programs. However, you can write programs that execute very rapidly in machine language compared to BASIC.

magnitude—the value or size of a number, without regard to its sign. For example +2, and -2 both have a magnitude of 2. The magnitude is also called the absolute value of a number (see *ABS**).

main program—portion of the program that is the main control.

mantissa[*]—the base number portion of a number expressed in *scientific notation**. In 3.264E + 4, the *mantissa* is 3.264.

mass storage device[*]—an *accessory device**, such as a cassette recorder or disk drive, which stores programs and/or *data** for later use by the computer. This information is usually recorded in a format readable by the computer, not people.

memory[*]—see *RAM*, and *ROM*, and *mass storage device*.

milliseconds—thousandths of a second, where the prefix milli means 1/1000. The abbreviation of milliseconds is msec. or just ms.

mode—manner or way.

module[*]—see *Command Module*.

msec.—abbreviation of *milliseconds**.

multi—prefix meaning multiple; that is, more than one.

natural logarithm function—see *LOG**.

negative duration—in a *CALL SOUND* statement* a negative duration specifies that any sound being generated by another statement should be stopped and the current *CALL SOUND* be executed.

nested loops—loops which lie entirely within one another. See also *loop**, *FOR**.

NEW—a BASIC command which deletes the program in memory, and also any variables.

NEXT—see *FOR**.

noise[*]—various sounds which can be used to produce interesting sound effects. A *noise*, rather than a tone, is generated by the *CALL SOUND subprogram** when a negative frequency value is specified (-1 through -8).

normalized radix representation—a common way of internally storing binary numbers in the computer. After a decimal number has been converted to binary, the most significant "1" of the *mantissa** is not stored. For more details, see *BASIC: Advanced Concepts*.

null string[*]—a *string** which contains no characters and has zero length.

NUM—short form of *NUMBER**.

NUMBER—long form of *NUM*. A BASIC function which automatically provides a line number when you are typing in program lines. For example:

NUM 100

will generate line numbers 100, 110, 120, . . . To stop this automatic line numbering, just press the ENTER key without typing anything after the number. You can also stop by press the FCTN and CLEAR keys. The general form of *NUM* is

NUM starting line, increment

where the first argument is the starting line number and the second argument is the increment of the lines. The default values are 100 for the starting line and 10 as the increment.

Number Mode[*]—the mode assumed by the computer when it is automatically generating *program line** numbers for entering or changing statements.

numeric string—a string of legal numeric symbols, which may consist of the numerals 0-9; the "+", "-", or "." signs, the letter "E", and a decimal point. For example,

"1"

"-2"

"-3.5E-3"

are all legal numeric strings. Legal numeric strings can be converted by the *VAL** function to a number. Likewise the *STR** function can convert a number to a string.

Leading and trailing blanks in a numeric string are ignored by VAL. For example
 PRINT VAL (" 1 ")

will print the number 1.

The two leading and two trailing spaces after the numeral 1 in the string " 1 " are ignored by VAL.

numeric variable—a variable whose value is a number.

OLD—a BASIC command used to read in a program from a storage device. For example, the direct command

```
OLD CS1
```

will instruct the computer to read in the program stored on cassette device 1. You will be prompted by the computer as to what to do. See also *SAVE**.

ON—a BASIC word used with either GOTO or GOSUB to direct execution depending on the value of a variable. The general form is

```
ON numeric-expression GOTO line number 1, line number 2, . . .
```

or

```
ON numeric-expression GOSUB line number 1, line number 2, . . .
```

The computer evaluates the numeric expression and truncates it to an integer. Then the computer uses that value as a pointer to direct execution to the list of line numbers. For example, if $X = 2$, the statement

```
20 ON X GOTO 40,60,80
```

will direct the computer to line 60 since that is the second line number in the list. The ON is an example of a *conditional branch** statement.

OPEN—a BASIC word which can be used in a direct command or statement to open a device such as a cassette tape unit, disk, RS-232C interface, etc. for access by a program. See also *OUTPUT**, *RELATIVE**, *SEQUENTIAL**, *INTERNAL**, and *FIXED**.

operator[*]—a symbol used in calculations (numeric operators) or in relationship comparisons (relational operators). The numeric operators are +, -, *, /, ^, . The relational operators are >, <, =, <=, >=, <>.

OPTION BASE—a BASIC word used in a statement to set the lower limit of array subscripts to one instead of zero. The argument of OPTION BASE is either "0" or "1". For example, the statement

```
10 OPTION BASE 1
```

sets the lower limit of arrays to 1. If there is no OPTION BASE in a program, the default value is zero for the lower limit of array subscripts.

outer loop—a loop which lies entirely outside of another.

output[*]—(noun) information supplied by the computer; (verb) the process of transferring information from the computer's memory onto a device, such as a screen, line printer, or *mass storage device**.

OUTPUT—a BASIC word used in an OPEN statement to specify that a file may only be written.

overflow[*]—the condition which occurs when a rounded value greater than 9.999999999999999E127 or less than -9.999999999999999E127 is entered or computed. When this happens, the value is replaced by the computer's limit, a warning is displayed, and the *program** continues.

parameter[*]—any of a set of values that determine or affect the output of a *statement** or *function**.

Pascal—a computer language designed by Niklaus Wirth and named in honor of the famous 17th century physicist and mathematician, Blaise Pascal. For more information on the history of computers and fundamental computer technology, see the book *Foundations of Computer Technology*.

period—the time between events. For example, if you borrow money, the period is the time between your monthly payments. If you deposit money in a bank, the period is the time interval between payments of interest to you. For a sound wave described by a sine wave, the period is the time between two adjacent peaks of the wave. The frequency is 1 divided by the period. So if the time between peaks is .001 second, the frequency is 1000 cycles or 1000 Hz (see *Hertz**).

permutation—an ordered arrangement of objects. For example, the numbers 1, 2, and 3 can be arranged in the following ways

```
123
132
213
231
312
321
```

Each of these arrangements is called a permutation of 1, 2, and 3. Note that the order in which the numbers are arranged makes each permutation unique.

picture elements—the dots or elements which make up a picture. Also called pixels.

pitch—the frequency of a sound wave.

pixels—contraction of the term *picture elements**.

planes—layers in which *sprites** can be defined. Sprites in one plane can appear to pass over or under other sprites depending on which plane the sprite is defined in.

pointer—a variable which keeps track or points to another variable or item of data. Your computer uses an internal pointer to keep track of the next item to read in a *DATA** statement. See also *READ**.

POS—a BASIC string function which returns the starting position of a search string in a data string. The general form of POS is

POS (data string, search string, starting position of search)

For example, the direct command

```
PRINT POS("ABCDE", "D", 1)
```

4

returns a value of 4 because the "D" is found at the fourth position from the left of "ABCDE".

position function—see *POS**.

power-of-ten—see *scientific notation**.

principal—the amount you own. For example, if you put \$1000 into a savings account, the \$1000 is called the principal. If you earn 10% interest per year, then after one year your principal is \$1000 + \$100 = \$1100.

PRINT—a BASIC word that prints the value of variables or the results of calculations. PRINT is also used to output data to files on storage devices such as cassette tape and disk. PRINT may be used as a direct command or in a statement.

printable—capable of producing a visible character.

print line[*]—a 28-position line used by the PRINT and DISPLAY statements.

program[*]—a set of statements which tell the computer how to perform a complete task.

program line[*]—a line containing a single *statement**. The maximum length of a program line is 112 characters*.

prompt[*]—a symbol (>) which marks the beginning of each *command** or *program line** you enter; a symbol or phrase that requests input from the user.

pseudo—a prefix which literally means false. See also *pseudo-random number**.

pseudocode—literally meaning "false code". Pseudocode consists of English-like statements that describe the program.

pseudorandom—see *pseudo-random number**. Another spelling of pseudo-random.

pseudo-random number[*]—a number produced by a definite set of calculations (algorithm) but which is sufficiently random to be considered as such for some particular purpose. A true random number is obtained entirely by chance.

radix-100[*]—a number system based on 100. See "Accuracy Information" (in the *TI-99/4A User's Reference Guide*) for information on number representation.

RAM[*]—random access memory; the main memory where program statements and *data** are temporarily stored during program *execution**. New programs and data can be read in, accessed, and changed in RAM. Data stored in RAM is erased whenever the power is turned off or BASIC is exited.

random access memory—see *RAM**.

random number—a number whose value is not known in advance. See also *pseudo-random number**, and *RND**.

RANDOMIZE—a function which starts the pseudo random numbers of RND at a different place in the sequence of possible pseudo-random numbers. RANDOMIZE is commonly used as a statement in games which use RND so that you won't get the same random numbers and thus play, each time the program is run. You may also include an argument with RANDOMIZE, called the *seed**, to start the pseudo random numbers off at a certain place in the sequence of numbers. Parentheses are not used for the argument of RANDOMIZE. For example:

```
RANDOMIZE 1
```

supplies a seed value of 1.

READ—a BASIC word used in statements to assign values from one or more *DATA** statements to variables. For example

```
10 READ A,B
```

```
20 DATA 5,10
```

When the computer executes line 10, it assigns the values A=5 and B=10. The computer keeps track of the next item to be read with an internal *pointer**. See also *RESTORE**.

read-only memory—see *ROM**.

record[*]—(noun) a collection of related data elements, such as an individual's payroll information or a student's test scores. A group of similar records, such as a company's payroll records, is called a *file**.

relation—sometimes called a relationship; a group of logical operators and operands. For example,

```
2=3
"AB">="A"
```

are both relations.

RELATIVE—a BASIC word used in an *OPEN** statement to tell the computer that random-access files are to be used. Random-access files are to be used with a disk, but not with cassette tape.

REM—a BASIC word used in statements to provide documentation or remarks about a program. Any text following the REM is not executed by the computer. For example,

```
10 REM PRINT 2+2
20 PRINT 3+3
```

the statement of line 10 prints nothing because the text after the REM is not executed by the computer. If you list the program, you can read the remarks.

renumbers—see *RESEQUENCE**.

RES—short form of *RESEQUENCE**.

resequence—see *RESEQUENCE**.

RESEQUENCE—long form of *RES**. The RESEQUENCE or RES command renumbers program lines. The general form is

```
RES starting line, increment
```

For example:

```
RES 100,10
```

will resequence a program so that the lowest line number is 100 and the following lines are 110, 120, 130, . . .

The *default value** of the increment is 10 and so the above command is equivalent to

```
RES 100
```

The default value of the starting line is 100 and so the command

```
RES ,30
```

will renumber lines as 100, 130, 160, . . .

reserved word[*]—in programming languages, a special word with a predefined meaning. A reserved word must be spelled correctly, appear in the proper order in a *statement** or *command**, and cannot be used as a *variable** name.

RESTORE—a BASIC word which resets the pointer of *READ** back to the first *DATA** item of a list. The optional argument of RESTORE enables you to specify the line number of the DATA statement you want to reset the pointer to. For example:

```
10 READ A
20 DATA 5,2
30 DATA 10
40 RESTORE
50 READ B
60 PRINT A;B
```

the above program assigns A=5 when line 10 is executed. Line 40 restores the pointer to the "5" in line 20 because no argument is specified. So line 50 will assign B=5. However, if line 40 was

```
40 RESTORE 30
```

the pointer is set to the first item in line 30 and so line 50 sets B=10. RESTORE may also be used with storage devices such as tape and disk. For more details, see the *TI-99/4A User's Reference Guide*.

RETURN—a BASIC word used in a statement to end a subroutine call. The *GOSUB** directs the computer to begin execution from a *subroutine** at a certain line. The RETURN statement directs the computer to return from the subroutine.

return-variable—the second argument of *CALL KEY**. The return-variable is assigned the value read from the keyboard by CALL KEY.

RND—a BASIC function which generates a *pseudo-random number**. No argument is needed for RND. Each time RND is used, it selects the next pseudorandom number from the thousands available. Every time you issue a NEW command, the sequence starts off with the same initial pseudorandom number of .5291877823. Likewise, the RUN, BYE, FCTN and QUIT commands, or turning off power to the computer will start the pseudorandom numbers from the beginning with .5291877823.

ROM[*]—read-only memory; certain instructions for the computer are permanently stored in ROM and can be accessed but cannot be changed. Turning the power off does not erase ROM.

RUN—a BASIC command which tells the computer to start executing the program in memory from its lowest line number. If you supply an argument to RUN, the computer will start execution from that line. For example:

```
RUN 30
```

will start execution from line 30. Note that the argument of RUN is not in parentheses.

Run Mode[*]—when the computer is *executing** a program, it is in Run Mode. Run Mode is terminated when program execution ends normally or abnormally. You can cause the computer to leave Run Mode by pressing CLEAR during program execution (see *Breakpoint**).

SAVE—a BASIC word used in a direct command used to write a program to a storage device. For example, the direct command

```
SAVE CS1
```

will instruct the computer to store the program in memory on cassette unit 1. You will be prompted by the computer as to how to do this. See also *OLD**.

saving programs—a way you can save programs on a storage material such as cassette tape, wafer tape, or disk. Tape and disk use magnetic materials for storage of information. Thus, when you turn the power off, your program has not disappeared if you've saved it. Data can also be saved (see *OPEN**). The command to save a program varies with the type of device used for storage. For example, with a cassette recorder, just type in

```
SAVE CS1
```

and press the ENTER key. Then follow the instructions shown by your computer. Consult the manual that came with your storage device for more details on how to use them. See also *load**.

scientific notation[*]—a method of expressing very large or very small numbers by using a base number (*mantissa**) times ten raised to some power (*exponent**). To represent scientific notation in TI BASIC, enter the sign, then the mantissa, the letter E, and the power of ten (preceded by a minus sign if negative). For example, 3.264E4; -2.47E-17.

screen color—see *CALL SCREEN**.

scroll[*]—to move the text on the screen so that additional information can be displayed.

scrolling—verb form of *scroll*. The contents of your display move up the top of the screen as new material is added to the bottom of the screen. So as new lines are added to the bottom, the older lines disappear from the top.

seed—the argument of *RANDOMIZE**.

SEG\$—a BASIC string function which returns a segment of a string. The general form of SEG\$ is

```
SEG$(string, starting position, final position)
```

where a portion or segment of the string is returned from the starting position to the final position. For example, the direct command

```
PRINT SEG$("ABCDE", 3, 5)
CDE
```

prints the segment "CDE" returned by the SEG\$ function.

segment function—see *SEG\$**.

sequential—occurring one after another.

SEQUENTIAL—a BASIC word used in an *OPEN** statement to tell the computer that sequential files are to be used. Sequential files must be used with cassette tape.

SGN—the sign function of BASIC. This function returns a value of 1 if its argument is greater than zero; a value of zero if the argument equals zero; or a value of -1 if the argument is less than zero. For example:

```
PRINT SGN(2.5)
1
PRINT SGN(0)
0
PRINT SGN(-2.5)
-1
```

sign function—see *SGN**.

simple variable—a variable which is not a dimensioned variable. A simple variable is accessed by its name and not by an *index number**. Simple variables are not declared in a *DIM** statement. See also *dimensioned variables**.

simulate—to act in all details like something else. An imitation does not act in all details like the original, but a simulation attempts to.

SIN—the sine of its argument in radians is returned by the SIN function. For a right triangle, the sine of an angle is the ratio of the side opposite the angle to the hypotenuse. For example:

```
PRINT SIN(.5)
```

.4794255386

sine function—see *SIN**.

software[*]—various programs which are executed by the computer, including programs built into the computer, *Command Module** programs, and programs entered by the user.

source code—the program that you want to execute. The BASIC *interpreter** is a program that executes every line of your source code (what you type in), line by line. In contrast, a compiler translates all the source code into *machine language** before executing the machine language equivalent of your high level source code.

sprites—graphic images maintained in special hardware.

SQR—the square root function. The computer calculates the square root of the argument of SQR. For example:

```
PRINT SQR(25)
5
```

gives a value of 5 because the square root of 25 is 5.

square root function—see *SQR**.

stack—special area of the computer's memory which is accessed in a sequential manner.

statement[*]—an instruction preceded by a line number in a program. IN TI BASIC, only one statement is allowed in a *program line**.

static—unchanging

status-variable—the third argument of *CALL KEY**. The status-variable returns a value of 1, -1, or 0 after the computer executes a *CALL KEY* statement. A value of 1 is returned if a new key was pressed since the last time *CALL KEY* was executed. A value of -1 is returned if the same key was pressed, while 0 is returned if no key was pressed.

STEP—see *FOR**.

STOP—a BASIC word which causes the program execution to end. Equivalent to *END**.

STR\$—a BASIC string function which converts a number or numeric expression to a string. For example,

```
PRINT STR$(100)
100
```

prints the string "100". The STR\$ and VAL functions are inverse functions, that is, for any string X\$, the following identity holds.

```
X$=STR$(VAL(X$))
```

string[*]—a series of letters, numbers, and symbols treated as a unit.

string function—see *STR\$** and *string functions**.

string functions—BASIC functions which apply to strings. For example, *CHR\$**, *ASC**, *LEN**, *POS**, *STR\$**, *VAL**, and *SEGS** are string functions.

subprogram[*]—a predefined general-purpose procedure accessible to the user through the *CALL* statement in TI BASIC. Subprograms extend the capability of BASIC and cannot be easily programmed in BASIC.

subroutine call—the process by which the computer calls a subroutine. The computer first stores the address of the next instruction it would have executed on the stack, and then starts executing from the line number following the *GOSUB**.

subroutines[*]—a program segment which can be used more than once during the *execution** of a program, such as a complex set of calculations or a print routine. In TI BASIC, a subroutine is entered by a *GOSUB* statement and ends with a *RETURN* statement.

subscript[*]—a numeric expression which specifies a particular item in an *array**. In TI BASIC the subscript is written in parentheses immediately following the array name.

TAB—the TAB function moves the current printing position to the column given by the argument of TAB. Columns are numbered from 1 to 28 where column 1 is the left most column. For example:

```
PRINT TAB(3);"A"
A
```

prints the letter "A" in the third column from the left because the argument of TAB is "3". The actual column at which tabbing starts also depends on the number of characters to be printed. See the discussion in the book.

tabbing—verb form of *TAB**.

TAN—the TAN function returns the tangent of its argument in radians. The tangent of a right triangle is defined as the side opposite the angle to the side adjacent the angle. For example:

```
PRINT TAN(.5)
.5463024898
```

tangent function—see *TAN**.

three-dimensional array—dimensioned variables with three index variables. For example, *N(I,J,K)*.

trace[*]—listing the order in which the computer performs program statements. Tracing the line numbers can help you find errors in a program flow.

TRACE—a BASIC word that can be used to trace the execution of a program. TRACE can be used as either a direct command or in a statement. When the program executes, the computer displays the line numbers being executed. See also *UNTRACE**.

transparent—clear, having no color of its own. A color code of "1" selects the transparent color. See also Appendix of Color Codes, *CALL COLOR**, and *CALL SCREEN**.

truncated—to cut off or truncate. For example, the *INT* function truncates the decimal part of a number.

two-dimensional array—dimensioned variables with two index variables. For example, *N(I,J)* is a two-dimensional array.

unary operator—an operator which acts on a single operand. For example, the unary minus or negation operator, inverts the sign of the following operator. An example is -2. The "-" is the unary minus. It makes the "+2" which follows into a "-2."

UNBREAK—a BASIC word which can be used in either a direct command or statement to remove effects of the *BREAK**.

unconditional jump—forcing the computer to start execution from another line. A *GOTO* is an example of an unconditional jump (see *GOTO**). An *IF-THEN-ELSE* statement is a *conditional jump** (also called a *conditional branch**), because the computer first tests the *IF** relation before deciding whether to jump.

underflow[*]—the condition which occurs when the computer generates a numeric value greater than -1E-128, less than 1E-128, and not zero. When an underflow occurs, the value is replaced by zero.

UNTRACE—a BASIC word which ends the action of *TRACE**. UNTRACE can be used as either a direct command or in a program statement.

user-defined function.—see *DEF**.

user-friendly—a program designed with consideration for the user, especially a new user.

VAL—a BASIC string function which converts a numeric string or numeric string expression to a number. For example:

```
PRINT VAL("100")
100
```

returns the number 100. See also *STRS**.

value function—see *VAL**.

variable[*]—a name given to a value which may vary during program execution. You can think of a variable as a memory location where values can be replaced by new values during program execution.

variable-length records[*]—records in a *file** which vary in length depending on the amount of *data** per *record**. Using variable-length records conserves space on a file. Variable-length records can only be accessed sequentially.

volatile—to disappear unless maintained. For example, the contents of *RAM** may disappear unless power is maintained to the computer. So *RAM* is said to be volatile. In contrast, *ROM** is non-volatile. That is why the BASIC in your computer is stored in ROM. The BASIC won't disappear when power is turned off because it's stored in ROM, unlike your programs which are stored in RAM. However, you can't change the ROM and so that's why programs are stored in RAM.

volume—the amplitude or loudness of a sound produced by *CALL SOUND**.

Index

absolute value function, ABS 29
address 97
algorithms 124
alphanumeric 2
arctangent function, ATN 34
argument 19
arithmetic expression 12
arithmetic operator 11
array 90
ASCII 73
assembly language 171
auto repeat 2
background color 163
base 10
binary number 26
bit 27
BREAKPOINT 68
BREAK 114
BYE, DELETE 171
byte 27
CALL CHAR 155
CALL CLEAR 137
CALL COLOR 163
CALL GCHAR 166
CALL HCHAR 142
CALL KEY 137
CALL SCREEN 162
CALL SOUND 148
CALL VCHAR 142
character function, CHR\$ 117
common logs 32
compiler 171
compound interest 62
computer animation 155
concatenation 17
conditional branch 74
CONTINUE, CON 68
control characters 117
control codes 117
control variable 80
cosine function, COS 33
cursor 1
database 89
datafiles 99
DATA 96
debugging 69
default value 40
DEF 134
delimiter 68
destructive cursor 146
dimensioned variables 89
dimension 92
DIM 92
display memory 166
dummy argument 135
duration 148
dynamic 166
editing 3
element 90
END 70
EOF 171
execute 35
exponential function, EXP 30
exponential notation 8
exponentiation 10
exponent 10
false code 113
field 15
flowchart 111
foreground color 163
format 86
FOR 80
frequency 148
GOSUB 129
GOTO 67
grid 142

- hertz 149
- hexadecimal 156
- Hz 149
- IF 74
- incremented 81
- index number 90
- index variable 80
- initial value 81
- inner loop 83
- INPUT 55
- input 2
- integer function, INT 27
- interpreter 170
- inverse function, ASC 118
- Individual Retirement Account (IRA) 63
- key-unit 138
- kilobyte 27
- length function, LEN 118
- LET 47
- limit 81
- linear array 104
- LIST 36
- literal 16
- load 63
- logical operators 71
- loop 69
- machine language 170
- magnitude 20
- main program 130
- milliseconds 148
- mode 138
- msec. 148
- multi 104
- natural logarithm function, LOG 31
- negative duration 149
- nested loops 83
- NEW 42
- NEXT 80
- normalized radix representation 27
- null string 68
- NUMBER 42
- numeric string 17
- numeric variable 48
- NUM 42
- Number Mode 43
- OLD 63
- OPEN 102
- OPTION BASE 107
- outer loop 83
- OUTPUT 103
- output 5
- parameter 82
- period 62
- permutation 127
- picture elements 156
- pitch 148
- pixels 156
- planes 171
- pointer 97
- position function, POS 119
- power-of-ten 8
- principal 60
- printable 2
- program 35
- prompt 1
- pseudocode 113
- pseudorandom 25
- pseudo 25
- Pascal 139
- random number 24
- random-access memory, RAM 35
- RANDOMIZE 25
- read-only memory, ROM 35
- READ 96
- relation 71
- REM 109
- renumber 40
- RESEQUENCE 40
- resequence 40
- reserved word 50
- RESTORE 96
- RES 40
- return-variable 138
- RETURN 129
- RND 24
- RUN 36
- SAVE 63
- saving programs 63
- scientific 8
- screen color 162
- scrolling 4

seed 26
segment function, SEG\$ 121
sequential 103
sign function, SGN 30
simple variable 90
simulate 24
sine function, SIN 32
sprites 171
square root function, SQR 21
stack 130
statement 35
static 166
status-variable 139
STEP 81
STOP 70
string function, STR\$ 120
string functions 117
string 16
subprogram 137
subroutine call 130
subroutine 129
subscript 90
tabbing 19
TAB 19
tangent function, TAN 34
three-dimensional array 104
TRACE 113
transparent 164
truncated 53
two-dimensional array 104
unary operator 11
UNBREAK 115
unconditional jump 69
UNTRACE 114
user-defined function. 134
user-friendly 94
value function 121
VAL 121
variable 47
volatile 38
volume 149

TEXAS INSTRUMENTS 99/4A BASIC GUIDE

If you want to learn about computers, this book and your TI-99/4A will teach you how to program in BASIC. You'll learn programming by working through this book with your computer. The clearly written, well organized text and programs will build up your knowledge of computers and their applications. You'll see how to use your computer for:

- Business—computing interest on savings, mortgage payments, and record keeping.
- Education—teach and improve children's knowledge of the alphabet, spelling, and arithmetic.
- Personal—quickly find telephone numbers, addresses or any other information with the practical Database Program. Sort names or other data alphabetically with your computer and throw away your old index cards.
- Games—video and educational games. Pilot the Space Shuttle to retrieve damaged satellites but watch out for the meteor storm.

You'll also learn how to use the graphics and sound capabilities of the TI-99/4A to create practical, original, and fun programs of your own. All of the programs in this book are carefully and clearly explained with you in mind. If you want an easy-to-read, yet thorough introduction to computers, this book is for you.

ABOUT THE AUTHOR

Joseph C. Giarratano received his Ph.D. in Physics from the University of Texas at Austin in 1974. He has worked in the field of medical physics with interests in medical applications of computers; for Bell Labs and American Bell on the design of computer-based products. Dr. Giarratano is the author of numerous other computer books: *Foundations of Computer Technology*, *Modern Computer Concepts*, *BASIC: Fundamental Concepts*, *BASIC: Advanced Concepts*, *Timex/Sinclair Users Guide—Vols. I and II*, *Timex/Sinclair 1000 Dictionary and Reference Guide*, and the *Timelost* books, including the best selling *Timelost: TI-99/4A* version.

Computext, Inc.
P. O. Box 50942
Indianapolis, IN 46250

\$9.95

ISBN 0-913847-00-3