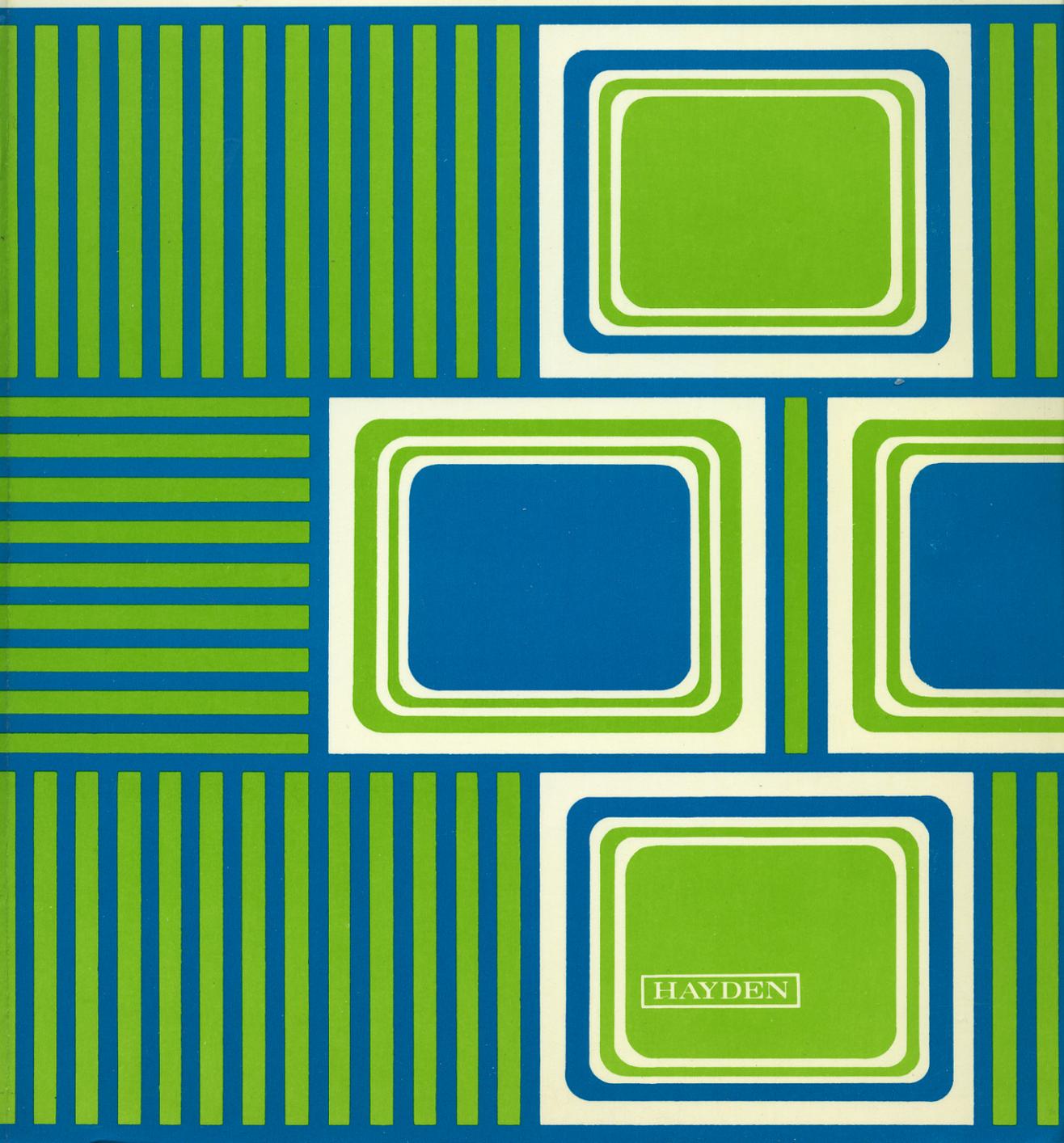


Don Inman, Ramon Zamora, and Bob Albrecht

TI BASIC

INTRODUCTION TO

TI BASIC



HAYDEN

**Introduction to
TI BASIC**

Introduction to TI BASIC

Don Inman
Ramon Zamora
Bob Albrecht



HAYDEN BOOK COMPANY, INC.
Hasbrouck Heights, New Jersey

Library of Congress Cataloging in Publication Data

Inman, Don.

Introduction to T1 BASIC.

Includes index.

1. Basic (Computer program language) 2. T1 99/4
(Computer)—Programming. I. Zamora, Ramon, joint
author. II. Albrecht, Robert L., joint author.

III. Title.

QA76.73.B3154 001.64'24 80-12825

ISBN 0-8104-5185-9

Portions of this work appeared originally in Texas Instruments' BEGINNER'S BASIC, copyright ©1979 by Texas Instruments Incorporated, and are used with the permission of the copyright holders.

Copyright © 1980 by HAYDEN BOOK COMPANY, INC. All rights reserved. No part of this book may be reprinted, or reproduced, or utilized in any form or by any electronic, mechanical, or other means, now known or hereafter invented, including photocopying and recording, or in any information storage and retrieval system, without permission in writing from the Publisher.

Printed in the United States of America

8 9 PRINTING

83 84 85 86 87 88 YEAR

Preface

This book is designed especially for those people with *beginning to intermediate* experience with the small home computer. In the early chapters, the authors assume that the readers are newcomers to the expanding world of personal computing. The material is presented slowly, and the users are led on a joyous and informative tour of the Texas Instruments 99/4 – an exciting new entry in the growing list of machines designed for use in the home. The readers are encouraged to examine the color, sound and graphics capabilities of the TI 99/4. The introduction of programming concepts and technical material is pursued within an overall context of exploration and discovery. New users will find that they can start using the TI 99/4 in the first chapter.

For the persons with intermediate skills, the later chapters contain a number of demonstrations of the machine's capabilities for use with larger programs. Animation on the screen, color graphics, sound, and music become the topics of discussion. Based on the material in the early parts of the book, the later chapters expand upon these seed ideas – providing the reader with the opportunity to see the same examples in a variety of situations. In the second half of the book, there is also a chapter on screen editing for the TI 99/4. This chapter shows the user how to access the various editing commands of the TI Home Computer – commands that facilitate making program changes and corrections.

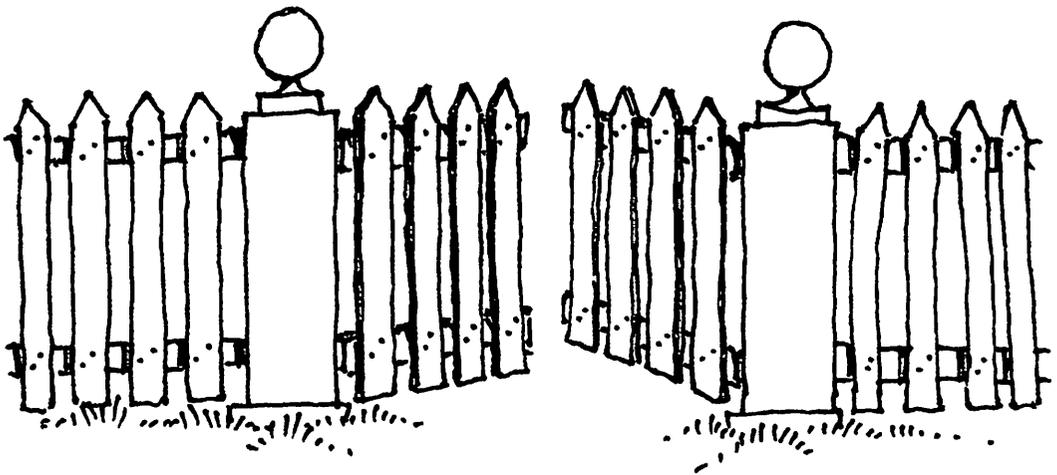
At the end of each chapter is a set of review questions. The questions and accompanying answers cover the high points of the material that has been presented. The reader can use the self-testing nature of the exercises to see just what has been learned, and to point out areas where more exploration may be needed. The exercises are designed to reinforce the learning process.

Do you need a computer in order to use the book? No! Although there is no complete substitute for sitting in front of a TI 99/4 as you read through this book, you can still use the book without a computer. In those cases where sound and color result, you will have to use your imaginations to try and “picture” what is being discussed. But for the most part, you can learn quite a lot about the TI Home Computer by working your way through the book. In fact for those persons who are “thinking” about getting a TI 99/4, this book is an excellent resource in helping them make that decision.

On the other hand, many readers of this book will have recently purchased a new TI Home Computer. In the box with the computer there is a booklet that was also written by the authors of this book. The material in the booklet and the present book overlap to some degree, but the book you are now reading goes well beyond the contents of the booklet. We have expanded each topic area, added new material, and, in general, designed a tour of your new machine that answers many of the questions you are likely to encounter on your adventure with your personal computer. The book you are holding will be invaluable as you make your journey into the rich land of the TI 99/4 – a land filled with many exciting hours of learning and recreation.

Chapter One is called the Gateway to Adventure. You are now approaching the Gateway – your adventure is about to begin. Congratulations to you for making this journey with us – a journey where there is much to learn, and much to enjoy! The gate appears to be opening . . . hurry and come along with us on this adventure.

*Don Inman
Ramon Zamora
Bob Albrecht*



Contents

Chapter One – Gateway to Adventure	1
Do It Now, 3	
The CALL CLEAR Statement, 5	
The IMMEDIATE Mode, 6	
The PRINT Statement, 6	
Correcting Mistakes, 8	
The LET Statement, 11	
Summary, 17	
Exercises, 18	
Answers, 19	
Chapter Two – Do It Now: Sound and Color Graphics	20
The CALL SOUND Statement, 20	
More Than One Tone, 22	
Noisy Sounds, 24	
Graphics, 25	
Summary, 32	
Exercises, 33	
Answers, 35	
Chapter Three – Simple Programming	36
Your First Program, 36	
Your Second Program, 38	
More About LISTing a Program, 44	
The INPUT Statement, 45	
Identifying the Answer, 49	
String Variables, 49	
Summary, 54	
Quick Review of Program Structure, 54	
Exercises, 55	
Answers, 57	
Chapter Four – Looping Sound and Color	58
The GO TO Statement 58	
GO TO With CALL SOUND, 60	
Loops, 62	
Musical Scales, 66	
A GO TO Loop with the CALL COLOR Statement, 68	
Error Messages, 73	
Summary, 75	
Exercises, 76	
Answers, 78	

Chapter Five – More Programming Power	80
The FOR-NEXT Statement, 80	
The FOR-NEXT Delay Loop, 83	
“Nested” FOR-NEXT Loops, 84	
Subroutines, 86	
Animation, 89	
Flashing Letters, 90	
Flashing Color Squares, 91	
Moving Color Squares, 92	
Error Conditions with FOR-NEXT, 93	
Summary, 94	
Exercises, 95	
Answers, 97	
Chapter Six – Beginning Simulation	98
The INTeger Function, 99	
The RND Function, 103	
A Two-Dice Simulation, 111	
Error Conditions with RND, 114	
Summary, 115	
Exercises, 116	
Answers, 118	
Chapter Seven – More Program Control Statements	119
The IF-THEN Statement, 119	
A Number Guessing Program, 123	
Random Notes, 127	
Musical Interlude, 129	
IF-THEN-ELSE Statement, 130	
The ON-GO TO Statement, 131	
The ON-GOSUB Statement, 133	
Summary, 134	
Exercises, 135	
Answers, 137	
Chapter Eight – Using Data Files	138
READ and DATA Statements, 138	
More Than One DATA Statement, 140	
The RESTORE Statement, 143	
READ a List, 143	
String DATA, 144	
Plain and Fancy PRINTing, 145	
Comma Spacing, 148	
Semicolon Spacing, 148	
The TAB Function, 150	
Rectangles and Squares, 153	
“Holes,” 157	
Summary, 158	
Exercises, 159	
Answers, 161	

Chapter Nine – One Dimensional Arrays	162
Tone Guessing Game, 165	
Color Organ, 166	
Customizing the Color Organ, 168	
Other Uses for Arrays, 170	
Calculating Salesmen’s Commissions, 172	
How To Run the Program, 175	
Exercises, 178	
Answers, 179	
Chapter Ten – Two Dimensions and Beyond	180
Three-Dimensional Arrays, 189	
Exercises, 198	
Answers, 200	
Chapter Eleven – Color, Graphics, Sound, and Animation	201
The CALL SCREEN Statement, 201	
Introducing CALL CHAR, 202	
COLOR with CHAR, 203	
PRINTed Patterns, 205	
Rectangles and Squares, 205	
“Triangular” Rectangles and Squares, 208	
Animation and Sound, 209	
Musical Interlude, 210	
The CALL CHAR Statement, 211	
A Block Figure with CALL CHAR, 216	
The Giant, 221	
Summary, 223	
Exercises, 224	
Answers, 226	
Chapter Twelve – More Strings	228
The Length of a String, 228	
Selecting a Substring of a String, 231	
How to Use the Program, 235	
Concatenating Strings, 236	
ASCII Codes, 239	
Finding a Character from its ASCII Code, 242	
Searching a String, 243	
Comparing Two Strings, 245	
Numbers to Strings and Strings to Numbers, 247	
Summary, 249	
Exercises, 250	
Answers, 252	
Chapter Thirteen – Editing	253
Editing More Than One Line, 256	
Deleting a Whole Line, 259	
Ignoring All Changes, 260	
Automatic Line Numbering, 261	
The RESequence Command, 262	

Summary, 264	
Exercises, 265	
Answers, 267	
Chapter Fourteen – Subroutines and Your Personal Library	268
The Delay Subroutine, 269	
Delay and Clear Subroutine, 270	
Putting a Border Around the Screen, 271	
Redefining a Block of Characters, 272	
Using VCHAR/HCHAR to Reward a Successful Guess, 272	
A Sound and Color Subroutine, 273	
Removing Spaces in a String, 274	
Removing Sets of Characters from a String, 276	
Rolling Dice, 277	
A Music Maker Subroutine, 278	
What’s Next in Adventureland?, 280	
Summary, 281	
Appendix A – Musical Notes and Frequencies	282
Appendix B – Character Codes	283
Appendix C – Color Codes	284
Appendix D – Mathematical Operations	285
Appendix E – Error Messages	297
Index	304

Introduction to TI BASIC

Chapter One

Gateway To Adventure

You are embarking on an adventure into the land of computers – a land filled with color, sound and visual effects. This book is your guide; with it you will learn how to use your Texas Instruments Home Computer. Even if you've never worked with a computer before, you can teach yourself, your family and friends to use, program and enjoy your TI Home Computer.

This book will help you to understand and use the computer language called BASIC. BASIC was developed in the middle 1960s at Dartmouth College by John Kemeny and Thomas Kurtz (Thanks, John and Tom!). BASIC is the most popular computer language, especially for beginners. BASIC is simple to learn, yet powerful enough to do most anything you want to do with computers.

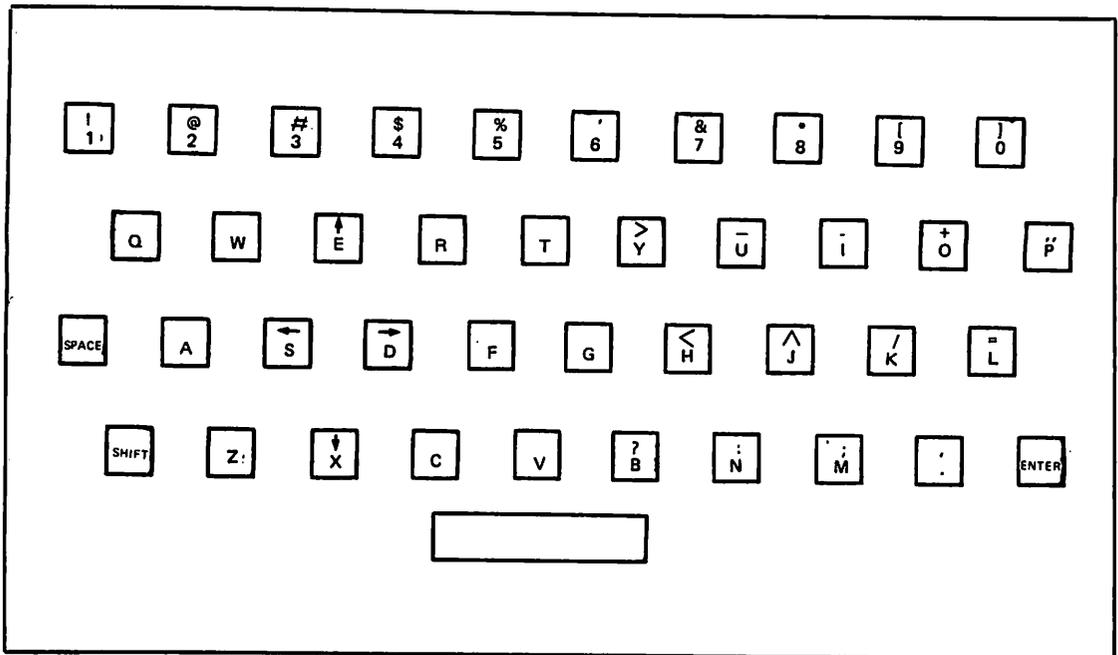
Learning to program is not mysterious. It is merely learning to communicate with the computer. You learn the language the computer understands so that you can “talk” to your Home Computer, telling it what to do and when to do it.

We assume that this is your first direct contact with a computer and that you want to use it immediately. Great! You will begin by learning interesting tricks the computer can perform. Then, you will learn to program it to successfully complete tasks for you, your family and your friends. You will find that you need little information to begin using your Home Computer and that you can quickly make use of its color, sound and graphical capabilities.

You stand at the Gateway to Adventure. Please enter; we will guide you as well as we can.

As you wend your way through this adventureland guide, try out the friendly encounters we provide for you and your computer. You will find that BASIC is much like English. You'll see words like PRINT, STOP and LIST. Most BASIC words and meanings are similar to the English words and definitions you already know. This feature makes BASIC easy to learn and fun to use.

Typing BASIC words into the computer is easy. Whether you are familiar with a typewriter or not, one look at your computer's keyboard tells the story. You see there the letters of the alphabet, punctuation marks, numbers, and other special characters.

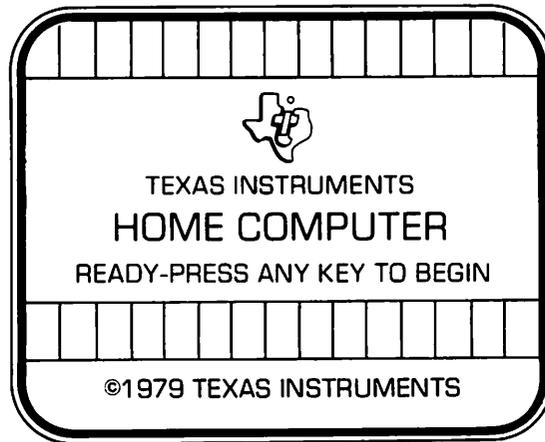


Everything you need is at your fingertips, right there on the keyboard. You type information on the keyboard. This information goes into the computer and is stored in the computer's *memory*. The information is also displayed on the TV screen so that you can see what you have typed. The computer uses the TV screen to display its responses to your instructions. It's easy! But, enough introduction – let's get started.

DO IT NOW

Plug-in and turn on your Home Computer. If you need help, consult the *User's Reference Guide* that came with your computer. Also, take a few minutes to review the operation of the keyboard. The *User's Reference Guide* contains a complete "key tour" of your Home Computer.

When you first turn on your Home Computer, the TV screen will look like this:



Press any key on the keyboard. The TV screen will then show the main *menu* of selections available to you.

**TEXAS INSTRUMENTS
HOME COMPUTER**

PRESS

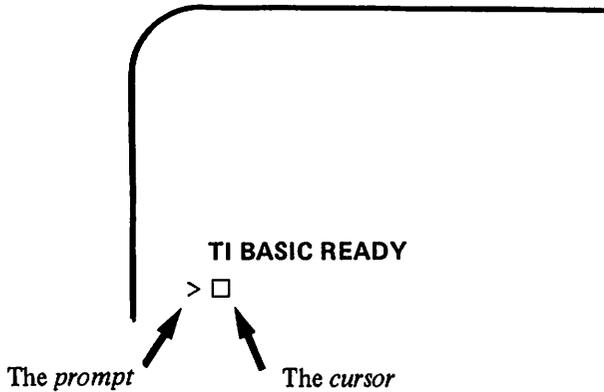
1 FOR TI BASIC

2 FOR EQUATION CALCULATOR

If a memory module is plugged into its slot on the console, its title will appear in the third position

In this book, we will talk only about option 1, TI BASIC. For information about other options, consult your friendly *User's Reference Guide*.

Since we want to use TI BASIC, let's do it! Press the `1` key to select TI BASIC. This is what you see:

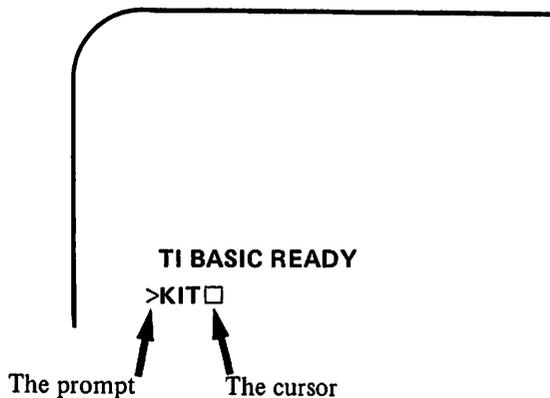


The computer is ready for you to use. The right pointing arrow without a tail (>) is called the *prompt*. The blinking rectangle (□) is called the *cursor*.

It is *your* turn to do something. The prompt (>) and the cursor (□) tell you that the computer is waiting patiently for *you* to type something on the keyboard.

If you don't type something on the keyboard, the computer will wait, and wait, and wait. Computers are very patient!

Type something. If you can't think of anything to type, type your name. Since we don't know your name, we will show you what happened when Kit typed her name.



Interesting. As Kit typed her name, the cursor moved to the right. The prompt didn't move. The prompt marks the *beginning* of a line at the left edge of the screen. The cursor tells you where you are now. The cursor marks the place where the next character you type on the keyboard will appear.

Kit has finished typing her name. *She* knows that she has finished, but the computer *doesn't* know that Kit has finished. After all, her name could be Kitty or Kitrinka or Kitterina.

To tell the computer that you are finished typing, press the **ENTER** key.

Kit, having finished typing, presses the **ENTER** key. This is what happens.

```

TI BASIC READY
>KIT
* INCORRECT STATEMENT
>□

```

Huh? How can Kit's name be an "INCORRECT STATEMENT"? After all, one does know one's name, doesn't one?

The computer should have said "I DON'T UNDERSTAND." That's what really happened. The computer did not understand the word "KIT" typed immediately after a prompt (>). The computer was waiting for a *statement*. A statement is an instruction to the computer, telling it to *do* something. A statement must begin with a special word, understood by the computer.

Yes, that is what this book is all about, those special words of TI BASIC, which the computer understands. So, get ready for your first special set of words.

The CALL CLEAR Statement

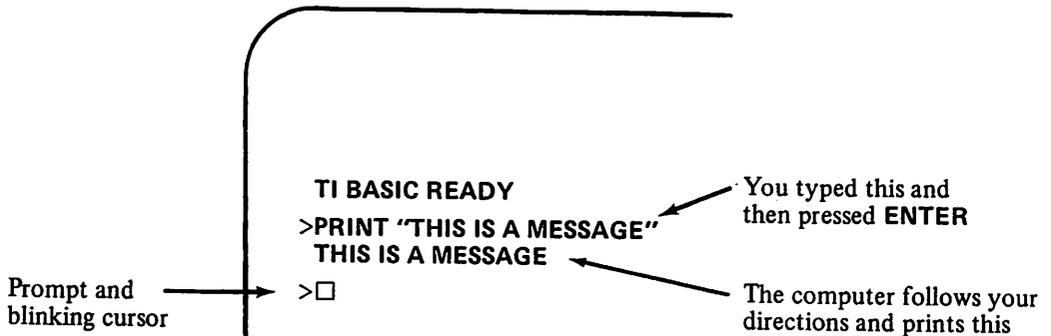
Occasionally, as you journey into the fantasyland of computers, you will encounter underbrush. You may notice clutter on the TV screen, especially if your fingers stumble as they seek the appropriate keys. If you want to clear the screen (get rid of the clutter), you can use the words CALL CLEAR.

```

CLUTTER CLUTTER
CLUTTER
CLUTTER
CLUTTER
>CALL CLEAR□ ← Type CALL CLEAR and
then press ENTER

```


After you have pressed the **ENTER** key, the display will look like this.



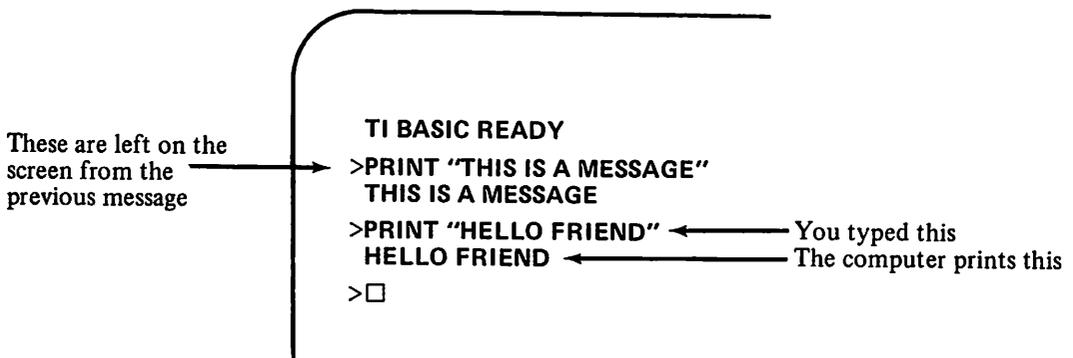
In our example, the *string*, enclosed in quotation marks in the PRINT statement, is a readable message. However, a string can be most any bunch of keyboard characters. The computer will print whatever you put between quotes in a PRINT statement.

The quotation marks tell the computer where the string begins and where it ends. The computer does *not* PRINT the quotation symbols, only the string *between* the marks.

Let's try another PRINT statement. Type this:

space space
 ↓ ↓
 PRINT "HELLO FRIEND"
 ↑ ↑
 Remember the quotes

Now press **ENTER**, and once again your computer does just what you tell it to do.



Did you notice the way the lines move up the screen when you press **ENTER** and again when the computer finishes printing the next line? This procedure is called "scrolling." Each time the computer prints a line, everything on the screen moves up one line to make room for the next line. The blinking cursor tells you that it's your turn and shows you where the next line will begin.

Press the **ENTER** key several times. Each time you press **ENTER**, the information on the screen will move up one line. If you keep pressing **ENTER**, you will eventually push information off the top of the screen, leaving only the prompt (>) along the left edge and the cursor (□) on the bottom line.

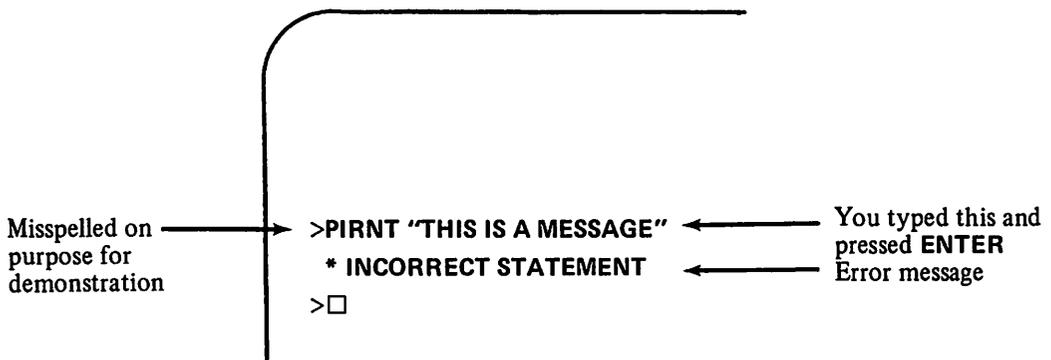
Correcting Mistakes

Every computer user makes typing mistakes. Here are some possible mistakes that your computer doesn't like in a **PRINT** statement.

- (1) Misspelling the word **PRINT**
- (2) A missing or extra quotation mark
- (3) Spaces in the word **PRINT**

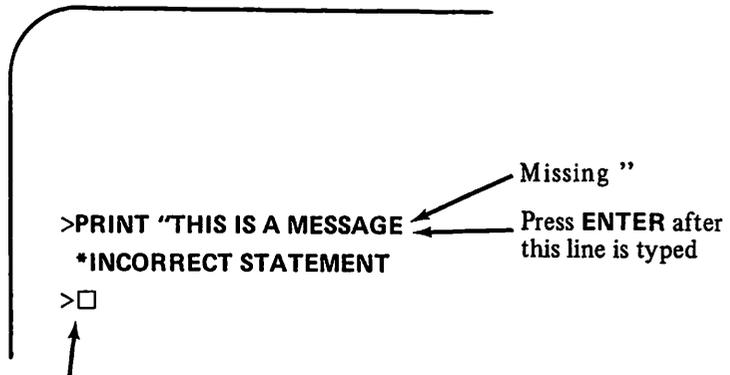
Experiment with intentional errors to familiarize yourself with error messages.

- (1) Misspelling the word **PRINT**



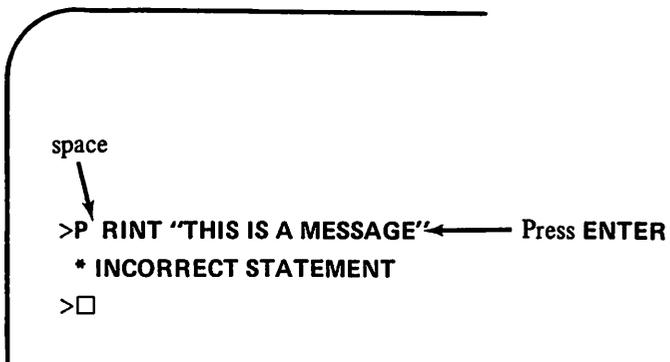
When your computer gives you an error message, examine the line that you typed to identify the error. Then retype the statement correctly.

(2) Missing or extra quotation marks.



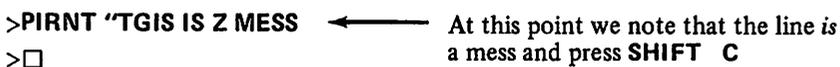
The prompt and cursor tell you that everything is OK. The computer is very patient — you can make as many mistakes as you wish. The computer will respond to each mistake and wait.

(3) Spaces in the word PRINT



There are several ways to correct typing errors *before* you press **ENTER**, and thus avoid error messages.

- (1) Holding down the **SHIFT** key and pressing **C** tells the computer to disregard what you've typed on the line. The incorrect line or partial line is not erased from the screen. It moves up one line, and the cursor moves to the next line so that you can start over.



- (2) If you spot the mistake just after you've made it, use the *backspace key* (hold down **SHIFT** and press the ← key) to move the cursor back to the error. Retype the line from that point on, and then press **ENTER**

PRINY□	←	Oops!
PRIN Y	←	We used backspace key.† Cursor is in position <i>over</i> the mistake.
PRINT□	←	We typed the correct letter.

- (3) If you finish typing a line and find a mistake near the beginning of the line, use the backspace key as previously shown, and retype the letter or word. Then, use the *forward key* (**SHIFT** →) to move the cursor to the end of the line. Then, press **ENTER**. Note that the forward key does *not* erase as it moves the cursor to the right. If you want to erase characters use the "space" key (space bar) to erase characters. Each space replaces a character.

>PIRNT "THIS IS A MESSAGE"□	Oops! Fortunately, we have not yet pressed ENTER .
>P I RNT "THIS IS A MESSAGE"	Using SHIFT ←, we position the cursor over the first mistake.
>PR I N T "THIS IS A MESSAGE"	We correct the mistake by pressing R and I .
>PRINT "THIS IS A MESSAGE"□	Using SHIFT →, we move the cursor to the end of the line.
>PRINT "THIS IS A MESSAGE" THIS IS A MESSAGE	We press ENTER . The computer typed this.
>□	

REMEMBER

- ← To move the cursor to the left, hold down the **SHIFT** key and press ← .
- To move the cursor to the right, hold down the **SHIFT** key and press → .

†Remember, hold down the **SHIFT** key and press ← .

The LET Statement

The LET statement assigns a *value* to a *variable*. A *variable* is the name of a place in the computer which can store (“remember”) a *number* or a *string*.

Variables come in two flavors, *numeric* variables and *string* variables. In this section, we will consider only numeric variables. The value of a numeric variable must be a number.

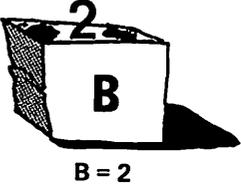
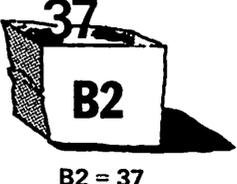
In the LET statement, the word LET is followed by the variable (the “name”), an equals sign (=), and finally, the numeric value that you’re assigning to the variable. Variables can be up to 15 characters long, but are generally kept short for typing convenience.

Let’s try a few examples. Type in the following lines. Press **ENTER** at the end of each line.

```
LET B = 2
LET B2 = 37
LET BETA = 123
```

You have just assigned *numeric* values to the *numeric* variables B, B2 and BETA. Or, you have just put numbers into the “boxes” B, B2 and BETA.

Examples:

BASIC STATEMENT	PICTORIAL
LET B = 2	
LET B2 = 37	
LET BETA = 123	

Of course, you can't see the "boxes." They are deep down inside the computer — too small to be seen by ordinary eyes.

To find out what is in a "box," use the PRINT statement

```
>PRINT B ← You type this and press ENTER
  2 ← Obedient as always, the computer prints the value of variable B
>□
```

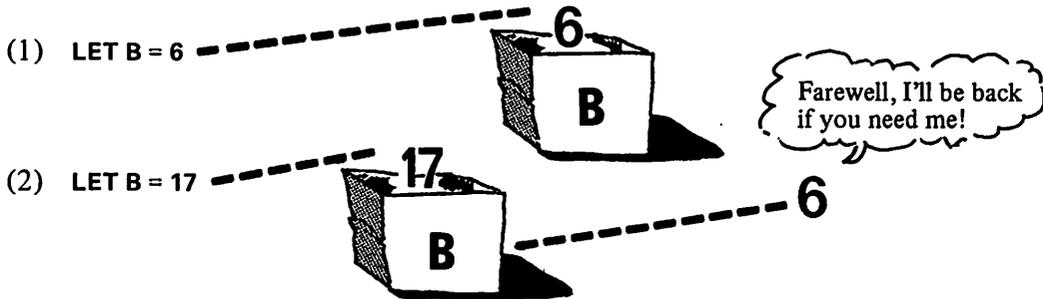
While we are at it, let's check boxes B2 and BETA.

```
>PRINT B2
  37
>PRINT BETA
  123
>□
```



Only one value at a time may be assigned to a given variable. However, at some time in the future, a new assignment may change the value.

Consider the following successive LET statements.



LET statement (1) assigned 6 as the value of B. Then, LET statement (2) assigned 17 as the value of B. The new value (17) replaced the old value (6) in box B. To convince yourself of this, do the following.

```
LET B = 6
LET B = 17
PRINT B
  17
```

← You type these statements

← The computer types the most recent value of B

From now on, we will usually omit the prompt (>) and the cursor (□)

Do you notice any differences between our PRINT statement above (PRINT B) and the PRINT statements that printed strings?

Above PRINT Statement: PRINT B
 Earlier PRINT Statement: PRINT "HELLO FRIEND"

The difference is . . . (suspense) . . . quotation marks!

The statement: `PRINT B`

tells the computer to print the *value* of the *variable* B.

The statement: `PRINT "HELLO THERE"`

tells the computer to print the string (HELLO THERE) which is between quotation marks, following the word PRINT. Try this.

```
LET KURT = 15 } ← You type these
PRINT KURT   }
15           ← The computer responds
```

Let's put strings and numeric variables together to say something about Kurt. But first, we want to remind you to use `CALL CLEAR` (and press `ENTER`) before you try out our examples. Down with clutter!

`CALL CLEAR` (and press `ENTER`)

The screen is clear, except for the prompt and the cursor, hovering expectantly near the bottom of the screen. Yes, it is *your* turn.

```
You type ↙
KURT = 15
PRINT "KURT IS" ; KURT
KURT IS 15
                ↘ Aha! A semicolon
```

The statement: `PRINT "KURT IS" ; KURT`

does not say KURT IS KURT although that is, indeed, very true. Instead, it says

- (1) Print the string: KURT IS
- (2) Then, print the *value* of the *variable*, KURT. Since the value of KURT is 15, our ever-obedient computer prints Kurt's age, which is 15.

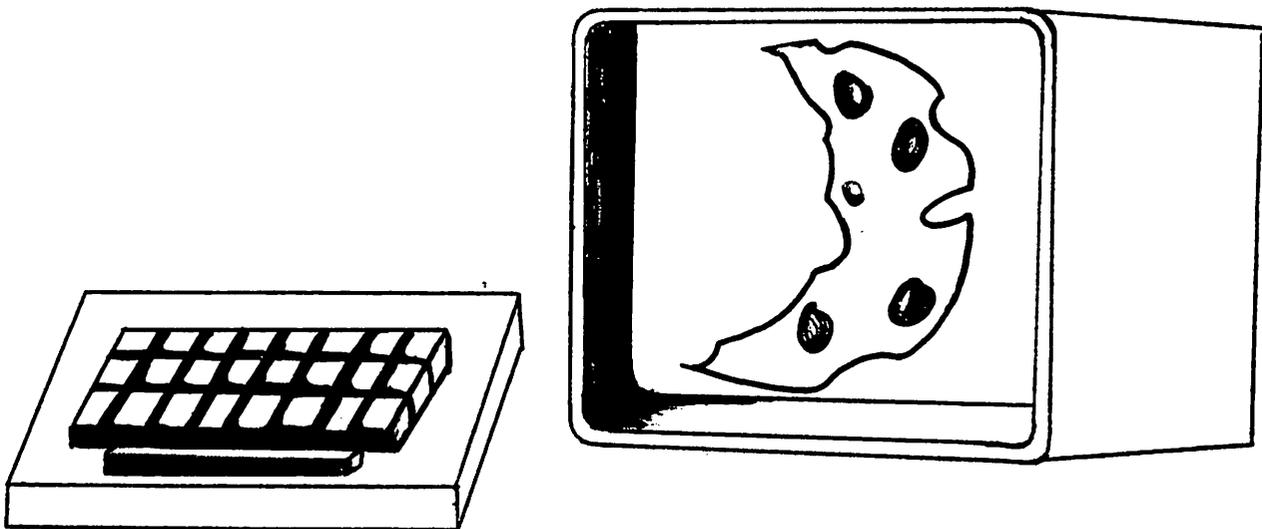
The semicolon (;) is a separator, or *delimiter* (as named by the High Priests of computer jargon). The semicolon separates one thing from another. How about another outrageous example?

`CALL CLEAR` (and press `ENTER`)

The screen is clear, ready for whatever we wish to do, uncluttered by our past efforts.

```
LET ANSWER = 7
PRINT "2 + 3 =" ; ANSWER
2 + 3 = 7
```

Now you know why your phone bill or gas and electric bill or department store bill is sometimes wrong. *Computers* do only what *people* tell them to do! If you tell a computer that the moon is made of blue cheese, the computer will believe it . . . and, perhaps tell it to everyone who "talks" to that computer.



Here is another look at the PRINT statement that lied to us.

```
PRINT "2 + 3 = " ; ANSWER
```

The above PRINT statement tells the computer to:

- (1) Print the string: 2 + 3 =
- (2) Then, print the value of ANSWER

If ANSWER had contained the correct answer, the computer would have been truthful. Let's teach our computer to be truthful.

METHOD 1

```
LET ANSWER = 5
PRINT "2 + 3 = " ; ANSWER
2 + 3 = 5
```

METHOD 2

```
LET ANSWER = 7
PRINT "2 + 3 = IS NOT" ; ANSWER
2 + 3 IS NOT
```

After variables are assigned values by LET statements, arithmetic operations can be performed on the variables within a PRINT statement

```

>LET A2 = 4
>LET A3 = 9
>PRINT A2 + A3 ; A3 - A2
  13   5
>□
    
```

Semicolon

Yes, of course, since A2 is 4 and A3 is 9, it is quite reasonable that $A2 + A3 = 13$ and $A3 - A2 = 5$. Our always clever computer picks up the *value A* of A2 and A3 adds these values in *expressions* such as $A2 + A3$ and $A3 - A2$.

A2 + A3
 ↙ This is an expression

You can also perform multiplication (*) and division (/) within expressions in a PRINT statement. The following PRINT statement is allowed in TI BASIC:

```
PRINT A2*A3; A3/A2
```

Instead of a semicolon, you may use a comma (,) as a separator, or delimiter (computer jargon) between items in a PRINT statement. Try this.

```
PRINT A2 + A3, A3 - A2
```

The screen now shows:

```

>LET A2 = 4
>LET A3 = 9
>PRINT A2 + A3 ; A3 - A2
  13   5
>PRINT A2 + A3 , A3 - A2
  13           5
>□
    
```

Answers are close together →

← Old stuff with a semicolon

← New line with a comma

Answers are far apart

The semicolon in a PRINT statement tells the computer to display the values close to each other. A comma in a PRINT statement tells the computer to place the values far apart. In fact, a comma tells the TI computer to place just two values on each line. Try this:

PRINT 1,2,3,4,5

The screen should now show

```

>LET A2 = 4
>LET A3 = 9
>PRINT A2 + A3 ; A3 - A2
  13  5
>PRINT A2 + A3 , A3 - A2
  13          5
>PRINT 1,2,3,4,5
  1          2
  3          4
  5
>□

```

If you were to use a semicolon between values in the last PRINT statement, all five values are displayed on the same line. Try it for yourself; enter the following

PRINT 1; 2; 3; 4; 5

Chapter Summary

Time to review what we have covered in this chapter and try the set of exercises that is provided. At the end of each chapter, you will find a set of exercises. Do as many of the exercises as you can. They help you review and remember what was talked about in the chapter. Exercise time!! 1-2-3-4; 1-2-3-4;...

- The prompt symbol is a right pointing caret (>)
- The cursor symbol is a blinking square (□)
- The computer executes a BASIC statement only after the **ENTER** key is pressed
- PRINT statements may include strings in quotes
- The shifted **C** key tells the computer to disregard the line just typed
- The shifted ← key moves the cursor left
- The shifted → key moves the cursor right
- The LET statement is used to assign values to variables. (The word LET is optional)
- When a comma is used as a separator in a PRINT statement, the results are printed far apart
- When a semicolon is used as a separator in a PRINT statement, the results are printed close together
- Arithmetic operations may be performed in a PRINT statement
- The CALL CLEAR statement clears the screen

(13) Pretend that you are the computer. Show what you will print.

```
>A = 7  
>B = 5  
>PRINT "A + B =" ; A + B
```

Answers to Chapter One Exercises

- (1) BASIC
- (2) keyboard
- (3) The TV screen, or video display, etc.
- (4) (a) cursor \square
(b) prompt $>$
- (5) YOU PASS THE TEST
- (6) scrolling
- (7) No quotation mark on the end of the PRINT STATEMENT
- (8) (a) moves the cursor to the left
(b) moves the cursor to the right
(c) tells the computer to disregard what you've typed on the current line (everything on the screen is scrolled up one line)
- (9) PRINT A;B
- (10) Yes
- (11) CALL CLEAR
- (12) PRINT A*B
- (13) A + B = 12

Chapter Two

Do It Now: Sound and Color Graphics

The CALL SOUND Statement

You can CALL upon your TI Home Computer to perform feats of musical magic. Simply use the CALL SOUND statement to play single notes, chords, or interesting “noise” through the audio portion of your TV monitor.

You can use the CALL SOUND statement to produce musical sounds over a range of several *octaves*, from 110 cycles per second (A below low C on a piano keyboard) to over 44,000 cycles per second, which is well above the limits of human hearing. Hmmmm... perhaps our computer can help us talk to dolphins.

Computer scientists electronics practitioners, audiophiles and others refer to “cycles per second” as “Hertz.”

1 cycle per second = 1 Hertz.

Perhaps you can guess why Hertz (named after Heinrich Rudolph Hertz, a German physicist) is shorter than *cycles per second*. However, in practice, even “Hertz” is abbreviated to “Hz.”

1 cycle per second = 1 Hertz = 1 Hz

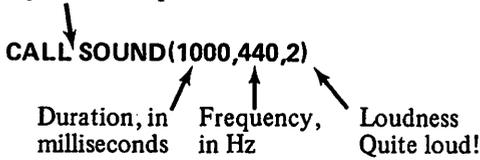
Your TI computer can make musical sounds ranging from 110 Hz to 44,000 Hz. Lousy bass but dynamite treble! You can also control the duration and the loudness of the sounds.

The duration can range from one millisecond to 4275 milliseconds. One thousand (1000) milliseconds equals one second, so the duration can range from 0.001 second to 4.275 seconds.

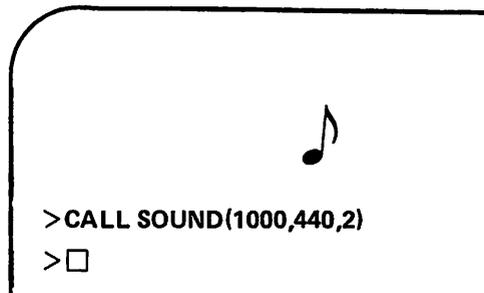
Loudness can be varied on a scale of 0 to 30. Zero (0) and one (1) produce the same sound levels and are the loudest. Thirty (30) produces the quietest sound. Remember, though, that the actual loudness level is ultimately determined by the volume control on the TV monitor.

Let's make music. Try this example.

Be sure you have a space here



OK, type the above CALL SOUND statement and press ENTER.



You should hear a tone of 440 Hz with a duration of 1000 milliseconds (one second) and a loudness value of 2 (quite loud!). Musicians call this tone "A above middle C."

Remember, the CALL SOUND statement looks like this:

CALL SOUND (duration, frequency, loudness)

Let's try another – we will check out the computer's lowest bass note.

CALL SOUND(500,110,15)

This time,

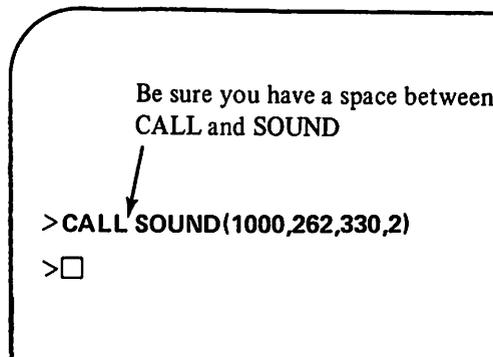
duration = 500 milliseconds = .5 seconds
 frequency = 110 Hz
 loudness = 15

We suggest that you try several CALL SOUND statements in order to explore the ranges for duration, frequency and loudness.

Remember these ranges:

duration: 1 to 4275 milliseconds
 frequency: 110 to 44000 Hz
 loudness: 0 or 1 (loud) to 30 (soft)

Type the above statement and press **ENTER**.



Because the statement above contains exactly 28 characters (letters, spaces, and symbols), the cursor moves down to the next line as soon as you type the close parenthesis symbol. Be sure that you remember to press **ENTER**.

Notice that you type the duration *parameter* (the number code that determines how long the sounds last) only one time – at the beginning of the CALL SOUND statement. The two sounds occur together for the same length of time. One the other hand, you *can* vary the loudness parameters. What would happen if you typed 5, instead of 2, for the second note's loudness? Try it!

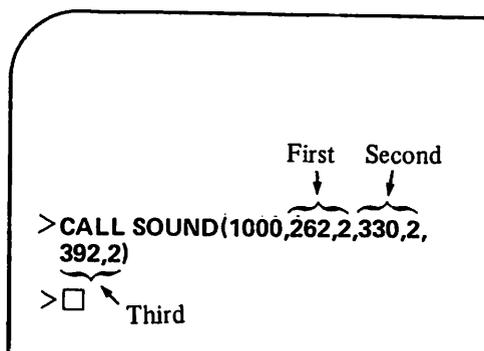
Remember, with *two* tones, CALL SOUND looks like this:

```
CALL SOUND(duration,frequency,loudness,frequency,loudness)
      ↑
      space
```

Next, try three tones

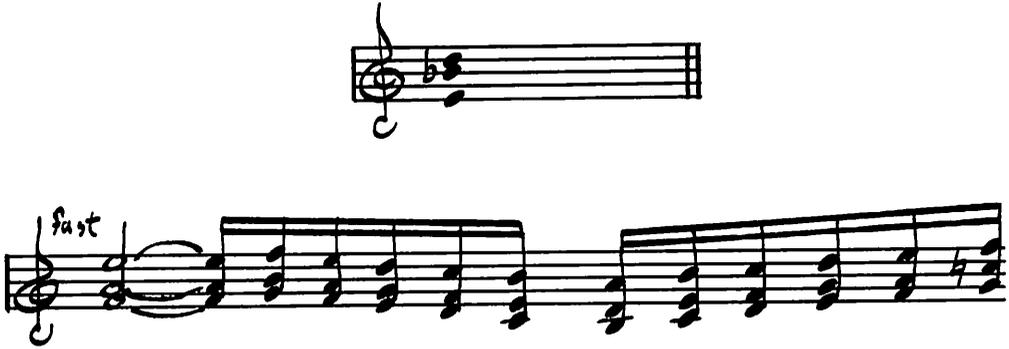
```
CALL SOUND(1000,262,2,330,2,312,2)
      ↗ ↘ ↗ ↘ ↗ ↘
    first second third
```

Type in the above statement and press **ENTER**.



Part of this CALL SOUND statement extends to the second line, since TI BASIC uses only 28 positions per line. This gives large, clear, readable text on the screen.

When you press **ENTER**, you will hear a three tone chord for one second.

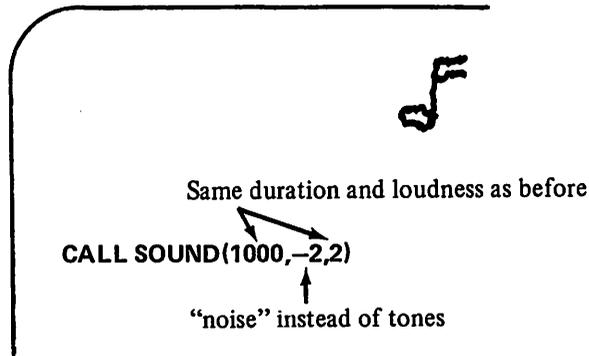


Noisy Sounds

You can also produce noise instead of music notes. Usually, we want to avoid noise when making music, but it may be useful at times. "Noise" is rather hard to define in words – it's best for you to experiment and hear for yourself what it sound like.

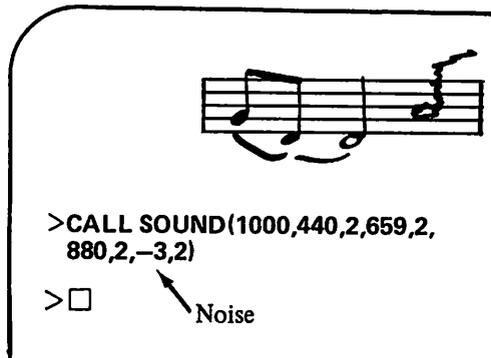
To produce a noise, use a negative integer from -1 to -8 as the "frequency." This will select one of eight possible built-in noises.

Try an example:

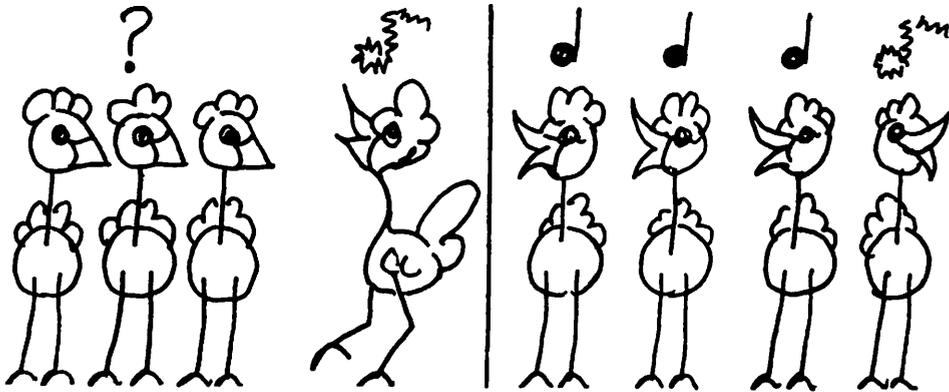


Explore the noises that your Home Computer can make. Try -1 , -2 and so on, down to -8 . Also try different durations and loudnesses.

You can simultaneously produce up to three tones and one "noise" over a given time duration.



In the above example, we used the same loudness (2) for all three tones and the noise. Experiment with other values for duration, frequency, loudness and noise within the required range of values for each. (A list of musical note frequencies is included in the Appendix.) You'll soon be able to create imaginative sound effects for use in your future programs. The IMMEDIATE Mode is helpful for this type of experimentation.

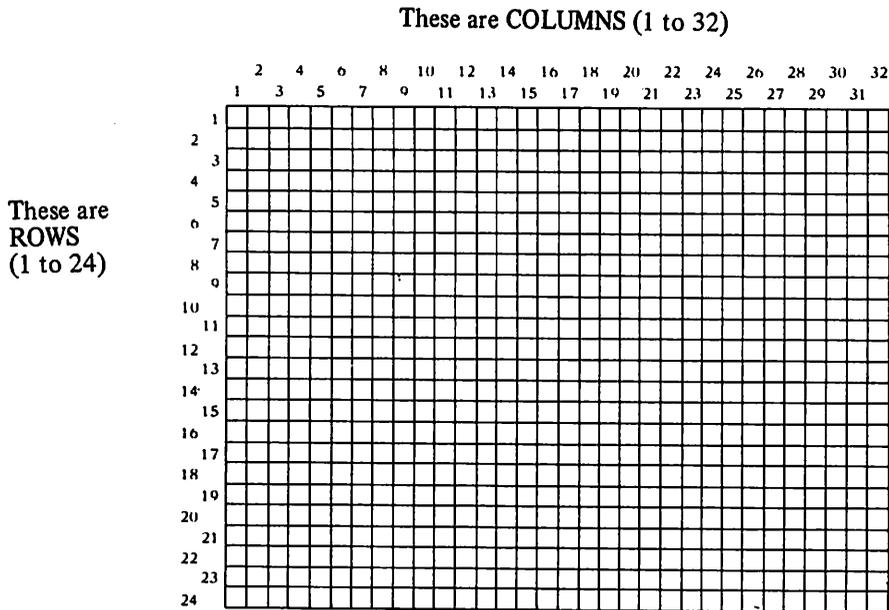


Graphics

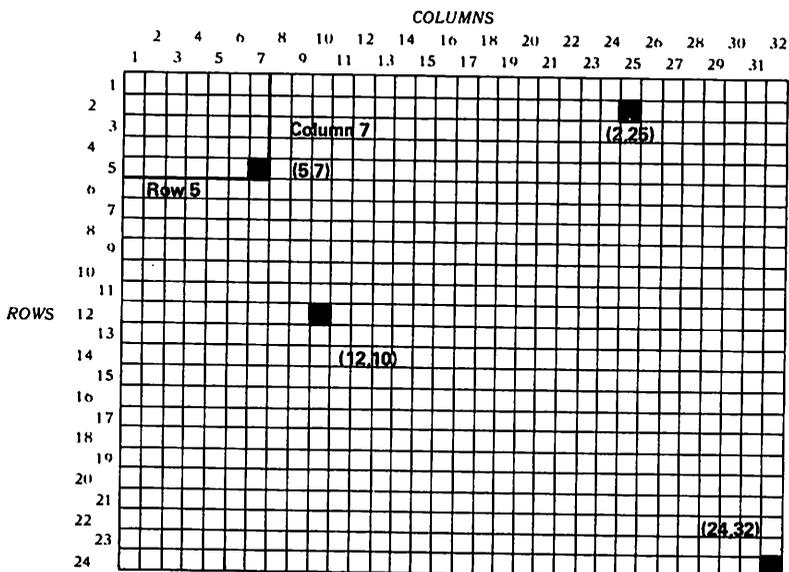
One of the most exciting things that you can do with your computer is to create colorful designs on the video display. This feature is one of the more significant advances for the home computer. Programs that might otherwise be dull come alive with the creative use of graphics. With your computer's capability, you can make a design, draw a picture, create a gameboard, draw a graph — let your imagination run wild.

In this section, we introduce two simple graphics statements, CALL VCHAR and CALL HCHAR. They position a character or draw a line of characters on the television screen.

A good way to begin is to think of the screen as a “grid” of square blocks, made up of 32 columns and 24 rows.



Each square on the grid is identified by two values (called coordinates) – a row number and a column number. For example, the coordinates 5,7 mean the fifth row and the seventh column. The coordinates 12, 10 mean the twelfth row and the tenth column. These are illustrated in the following diagram.



The next thing you need to know is how to place a character into a particular square on the screen.

For the time being, let's consider that a character is any one of 26 letters of the alphabet, the numbers 0 through 9, and certain other symbols, like the asterisk (*), the plus and minus signs (+ and -), and the slash mark (/). (Later, you'll learn how to "define" other characters for graphics.) Each character is assigned an identifying numeric value of its own, and the values for the full character set are given in the Appendix.

You can place a character in any spot you choose to by using either `CALL VCHAR` or `CALL HCHAR`, naming the two coordinates (row and column) and identifying the character by its numeric value.

Let's try a few examples. First, clear the screen. Type `CALL CLEAR` and press **ENTER**.

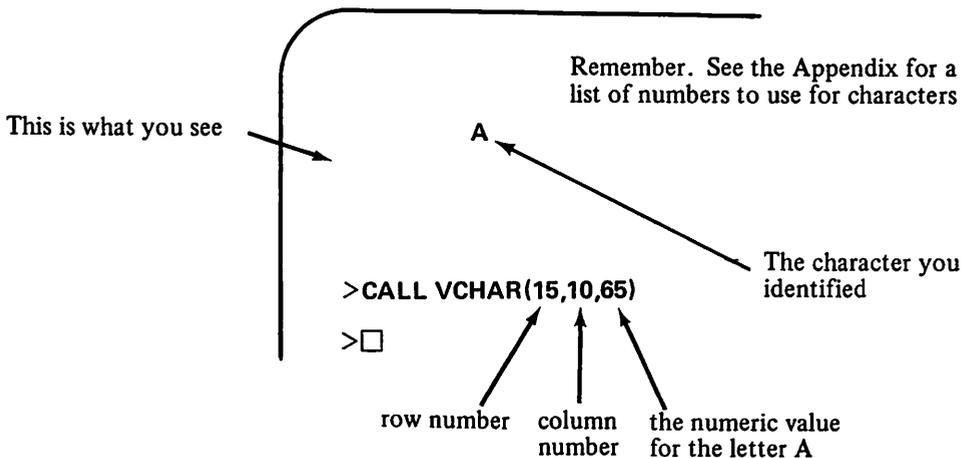
Now type:

`CALL VCHAR(15,10,65)` Check your typing and then press **ENTER**.

(Don't forget the parentheses in the statement – they're important!)

The parameters for this statement are:

15 = row number
10 = column number
65 = numerical value for the letter A



The character A is placed in the 15th row, 10th column. Try another one. Type:

`CALL VCHAR(1,3,66)` (and press **ENTER**)

This will put the character B (code 66) at the upper left corner of the screen (row 1, column 3). We call this location the “corner” even though the column is not at position 1. The reason for the “corner” being at this place is that some TV screens chop off columns 1,2,31 and 32. Characters printed in those columns do not appear on the screen.

One more time:

CALL VCHAR(1,30,67)


This time, the letter C (code 67) appears at the upper right “corner” of the screen in row 1, column 30.

Hmmm . . . have you noticed? The code for A is 65, the code for B is 66 and the code for C is 67. As you have probably guessed, the code for D is 68, the code for E is 69 and so on.

Remember, the general form for CALL VCHAR is as follows:

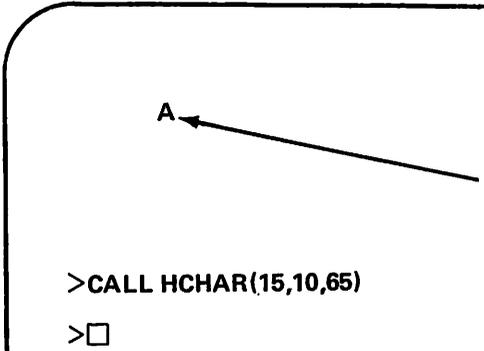
CALL VCHAR(row,column,code)

where *row* is an integer from 1 to 24, *column* is an integer from 1 to 32 and *code* is the numeric code for a character.

You can also place a character in a particular position by using the HCHAR statement. Let's try it!

Again, type CALL CLEAR and press ENTER to clear the screen. Then type:

CALL HCHAR(15,10,65)

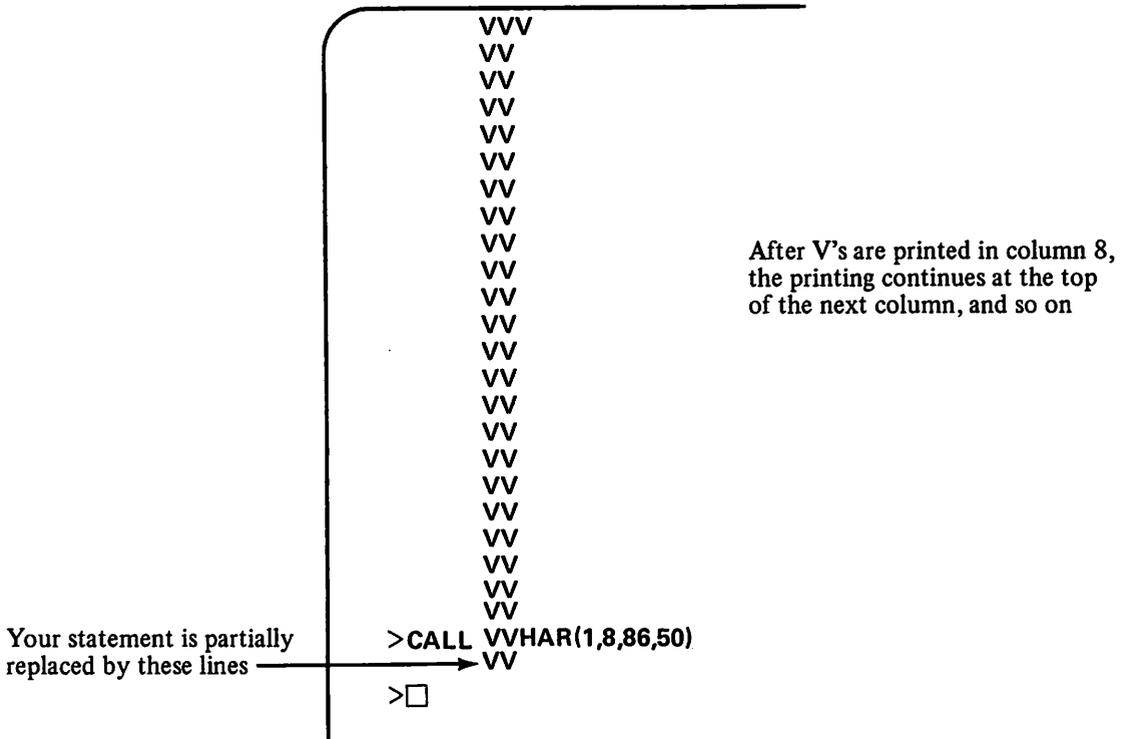


When you pressed ENTER an “A” appeared in row 15 column 10

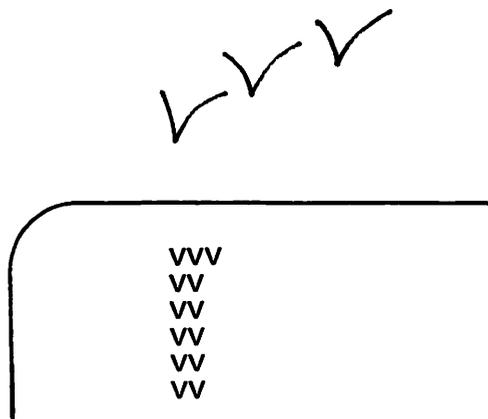
>CALL HCHAR(15,10,65)
>□

The entry order for the row number, the column number, and the character's numeric value is the same for both CALL VCHAR and CALL HCHAR. They both do the same thing *when you are positioning a single character on the screen.*

When you press **ENTER**, the screen should show:



(Note: Graphics in the Immediate Mode only are affected by the scrolling of the screen. That's why you don't actually see all 50 of the V's above – some have already scrolled off the top of the screen. In order to make room for the prompt and cursor, which are printed after the CALL VCHAR statement has been executed, everything on the screen is scrolled up on line. This pushes some of the V's off the screen's top.



For a big finale let's fill the screen with asterisks (numeric code 42). Type these lines, pressing **ENTER** at the end of each line.

CALL CLEAR

CALL HCHAR(1,1,42,768) 24 rows x 32 columns = 768 positions

Count the asterisks on your screen. How many do you see? (Don't forget some of them may have disappeared off the screen.)

Continue to experiment on your own. Try different characters (see the Appendix for the codes) and different positions. For example, can you fill the screen with your first-name initial?

Summary of Chapter Two

The summary concludes our "tour" in the IMMEDIATE MODE. In this chapter you've been introduced to these BASIC statements.

- **CALL SOUND**
- **CALL HCHAR**
- **CALL VCHAR**

Chapters One and Two have given you a slimpse of the BASIC language and your computer's capabilities. Try the chapter exercises, and then you're ready to get into the real fun—learning to program your computer with stored programs.

Chapter Two Exercises

- (1) The CALL SOUND statement controls three parameters. What are they?
 _____, _____, and _____
- (2) Two things control the volume of the sound resulting from the CALL SOUND statement. One is controlled by a parameter of the statement. What is the other?

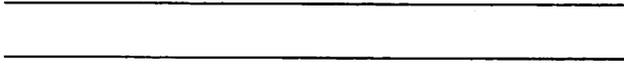
- (3) How many tones (notes) can be simultaneously played? _____
- (4) How many notes would be simultaneously played by the statement:
CALL SOUND(1000,440,2,659,2) ? _____
- (5) Can variables be used as parameters in CALL SOUND? _____
- (6) The graphics statements CALL VCHAR and CALL HCHAR use a grid of how many rows and how many columns?
 rows = _____ columns = _____
- (7) Give the row and column used in this statement:
CALL VCHAR(14,5,67) row = _____
 column = _____
- (8) Give the function of each of the parameters in this statement:
CALL VCHAR(5,8,72,20)
 5 = _____
 8 = _____
 72 = _____
 20 = _____
- (9) Describe the difference between the screen displays resulting from these two statements.
 (a) **CALL VCHAR(3,10,67,8)**
 (b) **CALL HCHAR(3,10,67,9)**
-
-

(10) Draw a sketch of 9(a) and 9(b).

9(a)

9(b)

(11) In one big finale, we used CALL HCHAR to fill the screen with asterisks. Redo the finale, using CALL VCHAR.

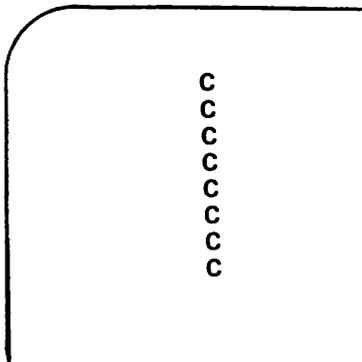


Answers to Chapter Two Exercises

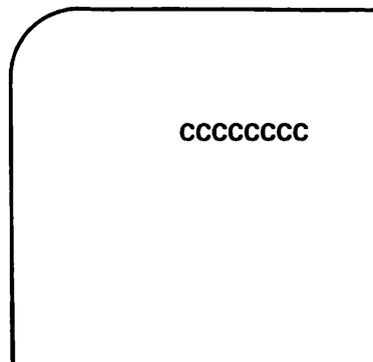
- (1) Frequency, duration, and volume (not necessarily in that order)
- (2) The volume control of the TV monitor
- (3) 3 (4 if you count noise)
- (4) Two
- (5) Yes
- (6) Rows = 24 Columns = 32 (or 28 if the TV chops off the first and last two positions)
- (7) Row = 14 Column = 5
- (8) 5 = row (of starting position)
8 = column (of starting position)
72 = value for character (H)
20 = number of repetitions (in the column)
- (9) (a) Causes a vertical column of 8 C's to be printed starting at row 3, column 10
(b) Causes a horizontal row of 9 C's to be printed starting at row 3, column 10

(10)

9(a)



9(b)



- (11) **CALL,CLEAR**
CALL VCHAR(1,1,42,768)

Chapter Three

Simple Programming

In Chapters One and Two you used the Immediate Mode statements to instruct the computer to do one thing at a time. Each statement was performed *immediately* after you pressed the **ENTER** key.

You typed: **PRINT "HI THERE!"** and pressed **ENTER**

The computer printed: **HI THERE!**

Now we will discuss *programs*, sets of statements that are *not* done immediately but are, instead, stored in the computer's memory. Then, when you are ready, you can tell the computer to "run" (or "execute") the program that is in its memory.

Your First Program

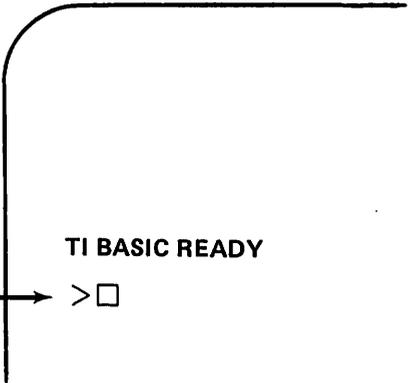
Let's begin by using a familiar friend, the **PRINT** statement, in your first program. Start by typing the word **NEW** and pressing **ENTER**. The word **NEW** is a command that tells the computer to erase any programs that may already be in its memory. **NEW** clears out the memory for a fresh start. We don't want the computer to be *confused* by having parts of two different programs.

Type **NEW** and press **ENTER**.

NEW clears the screen, erases any old program from the memory, and lets you know that it is *your* turn

TI BASIC READY

> □



Now type this program, pressing **ENTER** at the end of each program line:

```

space  space
  ↓    ↓
10 PRINT "NOW YOU LEARN"
20 PRINT "TO PROGRAM IN BASIC."
30 END

```

In computer terminology, you have just “entered” a program. Nothing to it! Each line was entered into memory when you pressed the **ENTER** key. **ENTER** is a signal to the computer that the line you typed is complete and ready to be stored.

If you have made *no* typing errors, the screen will look like this

```

TI BASIC READY
>10 PRINT "NOW YOU LEARN"
>20 PRINT "TO PROGRAM IN BASIC."
>30 END
>□

```

If there is an incorrect line, just retype that line correctly, *including the number at the beginning of the line*. You can make the corrections right there at the bottom of the screen. Then press **ENTER**. The computer will automatically replace the old line with the new, correct one.

*If you see an error before you press **ENTER** refer to Chapter One, for methods of correcting typing errors.*

Your program is displayed on the screen and is also stored in the computer’s memory. When you’re ready to see the program in action, type **CALL CLEAR** and press **ENTER**. The screen will be cleared, but your program won’t be erased – it’s still stored in the computer’s memory!

Now type **RUN** and press **ENTER** again.

Your first program just ran!!

```

>RUN
NOW YOU LEARN
TO PROGRAM IN BASIC.
** DONE **
>□

```

Want to “run” the program once more? Type RUN again and press **ENTER**.

```

> RUN
NOW YOU LEARN
TO PROGRAM IN BASIC.
** DONE **

> RUN
NOW YOU LEARN
TO PROGRAM IN BASIC.
** DONE **

> □

```

Each time you type RUN and press **ENTER**, the computer begins at the first statement and follows your instructions in order until it reaches the last statement. **END** means just what it says: the end, stop!

Did you notice that the display screen briefly turned green (if you’re using a color monitor) while the program was “running?” The screen turns green while a program is being executed and then changes back to its normal blue color when the program is “done.”

Your Second Program

As you enter your second program, we will spend time talking about some of the items and features you used to create your first program. One of the new things you may have noticed is that there was a number in front of each line in the program.

Home Computer BASIC is a line-oriented language. A BASIC program is a sequence of lines with each line having a unique number called a line number. The line number serves as a label for one particular line. When you type statements with line numbers, the computer does not execute them immediately. Instead, it stores them in its memory and waits patiently for you, the user, to tell it when to RUN (execute) the program. A valid line number is any decimal integer from 1 to 32767.

For your second program we will look at one that does a calculation, as well as printing. The program converts kilograms (K) to pounds ($2.2 * K$) and prints the answer.

```

10 LET K = 50
20 PRINT 2.2*K ← This is your second program
30 END

```

The preceding program consists of three *statements*.

```

10 LET K = 50 ← This is a statement
20 PRINT 2.2*K ← This is a statement
30 END ← This is a statement

```

Each statement begins with a *line* number.

```

Line number → 10 LET K = 50
Line number → 20 PRINT 2.2*K
Line number → 30 END

```

Notice that the lines are numbered in steps of ten (10). This is not required, but it allows you to insert additional lines between existing lines. In this way, you can avoid re-entering the whole program when an additional line must be inserted.

Program lines are executed in sequential order starting with the line having the smallest line number. The sequential execution continues until an error condition occurs, an END statement is executed, the sequential flow of the program is changed by one of several other BASIC statements, or you stop the program. All of these conditions will be discussed later, as they occur. For now, your programs will operate in sequential order.

When you entered and ran your first program you went through a series of steps:

- You typed the NEW command to make sure any old program in the computer was erased.
- You entered the program from the keyboard.
- You examined the program for any obvious entry errors. (If there were errors, you were free to correct them.)
- You typed RUN to run the program and produce the results.

Let's look at each of these considerations again, as you work with your second program. You begin by typing the command NEW. *Commands* are different from *statements*. They are not part of the program, and they do not have line numbers. They instruct the computer to do specific tasks immediately.

NEW	Instructs the computer to erase the program in its memory. (It also clears the screen, but don't confuse it with CALL CLEAR, which clears <i>only</i> the screen.)
RUN	Instructs the computer to perform (or "run") the program in its memory.

Now, enter the program to convert kilograms to pounds.

IMPORTANT NOTICE! If you make a mistake in typing in a line, simply retype it.

THIS IS WHAT YOU DO

- (1) Erase the old program.
Type: **NEW**

Before ENTER → >NEW□

Then press the **ENTER** key.

After ENTER → TI BASIC READY
>□

The cursor is now blinking at the position where your first line will begin.

- (2) Enter the new program.

Type: **10 LET K=50**

Before ENTER → TI BASIC READY
>10 LET K=50□ ← Cursor

Then press **ENTER**.

Line 10 is entered and its position on the screen moves up a line. The blinking cursor shows where the next line is to be entered.

After ENTER → TI BASIC READY
→ >10 LET K=50
Cursor → >□

THIS IS WHAT YOU DO

Type: **20 PRINT 2.2*K**

Then press **ENTER**

After **ENTER**

THIS IS WHAT YOU SEE

```

TI BASIC READY
>10 LET K=50
>20 PRINT 2.2*K
>□

```

Line 20 has been added to the program, and the cursor indicates the computer is ready for the next line.

Type: **30 END**

Then press **ENTER**

```

TI BASIC READY
>10 LET K=50
>20 PRINT 2.2*K
>30 END
>□

```

Our complete program has been entered. The cursor indicates that the computer is ready for another line or some new command.

Did you make any mistakes in typing? You can correct a mistake very easily; simply re-type the line in which the mistake occurred. Remember to begin with the line number!

(3) Examine the program for errors.

With your first program, you just looked at the screen to see if there were mistakes. With larger programs, it is helpful to be able to list, on the screen, the entire program or some of its parts. There is a command, called **LIST**, that helps you do this operation.

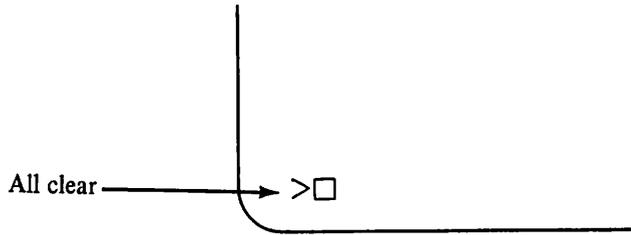
LIST Instructs the computer to show (or “list”) on the screen the program that is stored in its memory.

As you saw earlier, we use **NEW** *only* when we want to prepare the computer for storing a new program. Be careful in using **NEW**; when in doubt, use **LIST** first, so that you can examine the current program before erasing it.

LIST is a powerful aid when you are correcting or changing a program. It lets you get the program right on the screen in front of you, where you can check for or correct any errors.

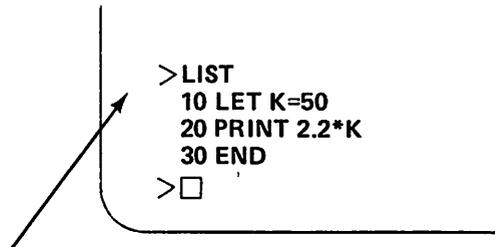
First, clear the screen, using an IMMEDIATE Mode statement.

Type: **CALL CLEAR**
Then press **ENTER**



Then tell the computer to LIST the program that is in its memory.

Type: **LIST**
Then press **ENTER**



Note: No line number is used with LIST. It is not part of the program. It is a command.

When you type LIST and press **ENTER**, the computer lists, on the screen, the program that is currently stored in its memory.

Before you RUN the program, check for typing errors. If there are any, retype the line correctly, *including the line number*, and press **ENTER**.

Let's mention two features of TI BASIC that may be slightly different from other versions of the language. First, a "prompting" character (to the left of the printing field on the screen) marks the start of every program line you type. You'll see its function more clearly when you begin to enter program lines that are longer than a single screen line. Second, the END statement in a program is optional in TI BASIC. Since it is a conventional part of BASIC, however, we used it in this example.

(4) RUN the program.

RUN Instructs the computer to perform (or "run") the program in its memory. RUN is a command.

Program lines are performed by the computer in numerical order. The computer first performs the line with the smallest line number, then proceeds to the next, then the next, and so on. Therefore, *you must be sure to number the lines of a program in the order you want the computer to follow.*

10 LET K=50 ← performed first
20 PRINT 2.2*K ← performed second
30 END ← performed last

The moment of truth has arrived! The most important, and one of the simplest, commands is given.

Type **RUN** and press **ENTER**.

```

>LIST
 10 LET K=50
 20 PRINT 2.2*K
 30 END
>RUN
 110
**DONE**
>□

```

The program →

Note again! RUN has no line number. It is a command for computer action, not a program statement

← The answer, 110 pounds

Computer says it's done

← Cursor waiting for another command

Your answer is on the screen: 50 kilograms is equal to 110 pounds. Suppose, however, that we want to find the number of pounds that are equivalent to 60 kilograms. Easy! We can do it by changing only one line – line 10.

Type: **10 LET K=60**

Press **ENTER**. Now type **RUN** and press **ENTER** again.

```

>LIST
 10 LET K=50
 20 PRINT 2.2*K
 30 END
>RUN
 110
**DONE**
>10 LET K=60
>RUN
 132
**DONE**
>□

```

← Your first answer, 2.2 x 50

← Your second answer, 2.2 x 60

You have now successfully entered your second program, RUN it, modified it, and RUN it again. Try some other modifications to line 10 before going on to the next section of the book.

More about LISTing a Program

Your second program is so short that it can be seen on the screen in its entirety. Programs that are longer than 24 lines will not fit on the screen but can be LISTed in sections. If a program does not produce the results expected when it is RUN, use the LIST command to re-examine the program. You may LIST an individual line, a group of sequential lines, as well as the whole program.

Getting back to your program example:

To LIST the entire program, type LIST and press ENTER.

```

>LIST
 10 LET K=60
 20 PRINT 2.2*K
 30 END
>□

```

← The entire program

To see a given line, type LIST followed by the line number.

```

>LIST
 10 LET K=60
 20 PRINT 2.2*K
 30 END
>LIST 20
 20 PRINT 2.2*K
>□

```

← Only one line

You can also list all the lines from a starting line number to an ending line number.

```

>LIST
 10 LET K=60
 20 PRINT 2.2*K
 30 END
>LIST 20
 20 PRINT 2.2*K
>LIST 20-30
 20 PRINT 2.2*K
 30 END
>□

```

← A group of lines is listed

Our kilograms to pounds program ends with an END statement. This statement is optional and can be omitted. To prove it, we will remove line 30 (but *only* line 30) from the program.

Clear the screen and list the program.

```

>LIST
 10 LET K=60
 20 PRINT 2.2*K
 30 END
>□

```

Delete line 30. To do this, simply type 30 and press **ENTER**.

```

>30
>□

```

List the program again.

```

>LIST
 10 LET K=60
 20 PRINT 2.2*K
>□

```

← Line 30 is gone

Remember: To delete a line from a stored program, type its line number and press **ENTER**.

Will the program run as it did before you deleted line 30? Find out. Type **RUN** and press **ENTER**.

The INPUT Statement

Here is our original kilogram program again.

```

10 LET K=50
20 PRINT 2.2*K
30 END

```

Line 10 assigns the value 50 to the variable K. We are using K to stand for *kilograms*. Then, line 20 tells the computer to multiply K by 2.2, thus computing the number of pounds corresponding to K kilograms. Line 20 also causes the computer to print the answer, which is the value 110.

We could, if we wish, retype line 10 and assign a different value to K, then rerun the program to get the answer for the new value of K. But there is a better way!

Here is your old program with an INPUT statement replacing line 10.

```
10 INPUT K ← INPUT statement
20 PRINT 2.2*K
30 END
```

The INPUT statement causes the computer to type a question mark, turn on the cursor and wait. It will wait until someone types a value and presses **ENTER**. The value is then assigned to the variable in the INPUT statement.

Let's demonstrate.

- (1) Type: NEW
- (2) Type in the program
- (3) Type: RUN

```
TI BASIC READY
>10 INPUT K
>20 PRINT 2.2*K
>30 END
>RUN
?□ ← The computer waits for you
to key in the value for K
```

- (4) Type: 60 and press **ENTER**

```
TI BASIC READY
>10 INPUT K
>20 PRINT 2.2*K
>30 END
>RUN
? 60 ← You typed 60
  132 ← It typed 132
**DONE**
>□ ← Ready for more work
```

RUN this program several times, experimenting with different values for the INPUT variable, K.

Remember, to use this program do the following:

- ➔ Type **RUN** and press **ENTER**.
The computer will show a question mark and the blinking cursor.
- ➔ Type a number and press **ENTER**.
The computer will assign your number as the value of K, compute and print the value of $2.2 * K$, and then stop.

When running programs that call for an INPUT, it is often confusing to see the question mark just appear on the screen. It may be difficult to remember what the computer is asking for. To help inform the user of the needed information, a different form of the INPUT statement is used. For example,

```
10 INPUT "KILOGRAMS=" : K
```

↑ string in quotes
 ↑ colon
 ↑ variable

Now try this new line 10 in place of the one in your previous example. Type in the line, then clear the screen and LIST the modified program.

Type: 10 INPUT "KILOGRAMS=":K
 Press ENTER.
 Type: CALL CLEAR
 Press ENTER.
 Type: LIST
 Press ENTER.

```
>LIST
10 INPUT "KILOGRAMS=" : K
20 PRINT 2.2*K
30 END
>□
```

Now, RUN the program.

Type: RUN
 Press ENTER.

```
>LIST
10 INPUT "KILOGRAMS=" : K
20 PRINT 2.2*K
30 END
>RUN
KILOGRAMS=□
```

Let's try 75 as our value.

Type: 75
 Press ENTER.

The answer

Ready for more work

```
>LIST
10 INPUT "KILOGRAMS=" : K
20 PRINT 2.2*K
30 END
>RUN
KILOGRAMS= 75
165
**DONE**
>□
```

Another change. Add a **CALL CLEAR** at the beginning of the program. To do this, we can use a line number less than 10. Let's use 5 as the line number.

Add this statement: **5 CALL CLEAR**

LIST the program. It should now look like this:

```

>LIST
5 CALL CLEAR ← Here is the new statement
10 INPUT "KILOGRAMS=":K
20 PRINT 2.2*K
30 END
>□

```

Now, when we **RUN** the program the computer will clear the screen, then ask: **KILOGRAMS=**.

Try it. **RUN** the program, then use 82 as the value entered.

Type: **RUN**
Press **ENTER**.

```

KILOGRAMS= □

```

Type: **82**
Press **ENTER**.

```

KILOGRAMS= 82
180.4
**DONE**
>□

```

The **CALL CLEAR** statement clears the screen at the beginning of the program execution. In this way, only the messages and input data used during the program **RUN** appear on the screen. This really **CLEARs** things up!

Identifying the Answer

Just as it was possible to attach the message "KILOGRAMS=?" to the INPUT statement, words and messages can be added to PRINT statements. In your current program the answer simply appears on the screen as a number. Another form of the PRINT statement can be used to identify what the answer represents. For example,

```
20 PRINT 2.2*K ; "POUNDS"
```

Answer ↑ ↑ ↑ Message
Semicolon to keep
answer and message
on the same line

Replace line 20 with the one shown above and RUN the program. If you do so, your screen shows the following information when you again use 82 for the number of kilograms.

```
KILOGRAMS= 82
180.4 POUNDS ← Answer has message attached
**DONE**
>□
```

If you have made all the changes, the program now looks like this:

```
5 CALL CLEAR
10 INPUT "KILOGRAMS= ":K
20 PRINT 2.2*K ; "POUNDS"
30 END
```

We suggest that you type LIST and press ENTER, in order to verify that the program stored in the computer's memory does look like the above.

String Variables

You already know what *numeric variables* are. A numeric variable can have a number as a value.

```
LET K = 50
```

↑ ↑
numeric numeric
variable value

A *string variable* is a variable which can have a *string* as a value. String variables differ from numeric variables in these ways:

A string variable name *must* end with a \$ (SHIFT \$).

The alphanumeric characters in the "string" *must* be enclosed in quotation marks.

"Strings" of numbers cannot have arithmetic operations performed with or upon them.

For example,

```
LET N$ = "JACK SPRAT"
```

↑ string

↑ string value, enclosed in quotation marks

These are also *string* variables: B\$ B2\$ BETA\$

But these are numeric variables: B B2 BETA

Let's try a couple of examples, using string variables in immediate mode.

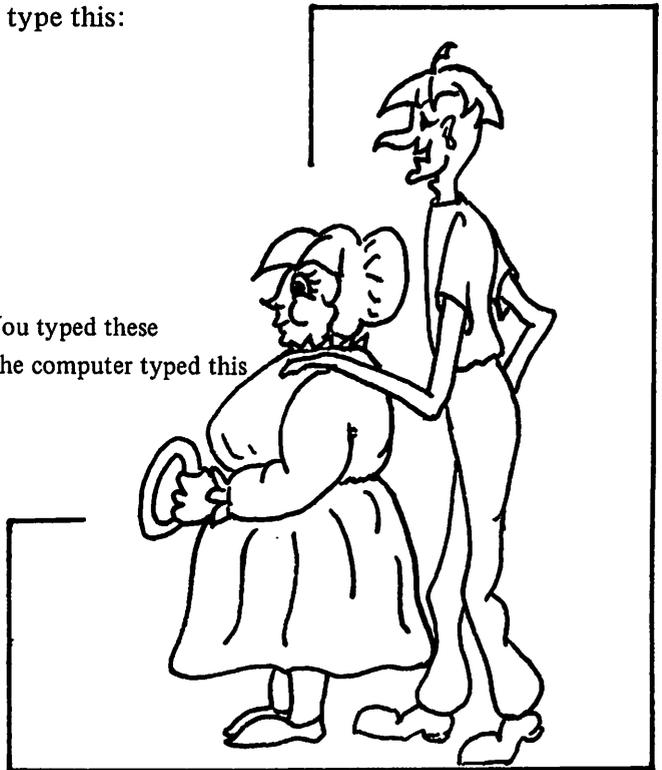
Clear the screen (CALL CLEAR) and type this:

```
LET N$="JACK SPRAT"
PRINT N$
```

```
>LET N$="JACK SPRAT"
>PRINT N$
JACK SPRAT
>□
```

← You typed these

← The computer typed this



Now type:

```
LET W$=" ATE NO FAT."
PRINT N$; W$
```

space

↓

↑ semicolon

Now let's look at them together:

```

>LET N$="JACK SPRAT"
>PRINT N$
  JACK SPRAT
>LET W$=" ATE NO FAT."
>PRINT N$;W$
  JACK SPRAT ATE NO FAT.
>□

```

With your last entry,
the computer typed this

Now back to your conversion program again. Your program now appears as follows:

```

5 CALL CLEAR
10 INPUT "KILOGRAMS=":K
20 PRINT 2.2*K;" POUNDS"
30 END

```

You can make your program a little more personal by adding a "string" variable in an INPUT statement. A string is just what the name implies — a string of alphanumeric characters that can be either a word or a set of words that form a message.

An INPUT statement can request an alphanumeric (letters and numbers) string, such as the user's name, for example, and assign that string to a string variable for further use in the program. A dollar sign, \$, must follow the variable name to distinguish it from a simple numeric variable.

For example,

```

8 INPUT "WHAT IS YOUR NAME" : B$

```

This string tells the user
what is desired

This variable is assigned
the value that you type

If you insert the line shown above along with the following lines into your program, your Home Computer starts "talking" to you.

```

15 CALL CLEAR
17 PRINT "OK,";B$
18 PRINT "HERE IS YOUR ANSWER! :"

```

semicolon

Remember when you see the new lines they will be *inserted* into your program in the proper places as determined by their line number.

After entering the lines, LIST the program to verify that this is true.

Type: LIST
Press ENTER.

```

>LIST
5 CALL CLEAR
8 INPUT "WHAT IS YOUR NAME":B$
10 INPUT "KILOGRAMS=":K
15 CALL CLEAR
17 PRINT "OK, ";B$
18 PRINT "HERE IS YOUR ANSWER!"
20 PRINT 2.2*K;" POUNDS"
30 END
>□

```

RUN the program.

Type RUN
Press ENTER.

```

WHAT IS YOUR NAME □

```

The program first asks for your name

Type your name.
Press ENTER.

Before ENTER

```

WHAT IS YOUR NAME GILGARA □

```

Your name typed here

After ENTER

```

WHAT IS YOUR NAME GILGARA
KILOGRAMS= □

```

If you now enter 82 for the kilogram request and hit ENTER the screen now shows:

The program remembered your name

```

OK, GILGARA
HERE IS YOUR ANSWER!
180.4 POUNDS
**DONE**
>□

```

Run the program several times and try using several different names. Type in your address when it asks for your name. Try your telephone number. The computer will call you anything you type.

Now let's use a string assignment statement: a LET statement. Type the following lines:

```
6 LET C$="HERE IS YOUR ANSWER!"  This is an insert line
18 PRINT C$                       This changes line 18
```

LIST the program and see if your program now looks like this:

```
>LIST
5 CALL CLEAR
6 LET C$="HERE IS YOUR ANSWER!" ← The insert
8 INPUT "WHAT IS YOUR NAME":B$
10 INPUT "KILOGRAMS=":K
15 CALL CLEAR
17 PRINT "OK, ";B$
18 PRINT C$ ← The change
20 PRINT 2.2*K;" POUNDS"
30 END
>□
```

When this version of the program is RUN, you would see the same display on the screen at each stage as in the last RUN. The string feature used in line 6 can be helpful if a message or part of a message is used several different places in the program. You just print the variable C\$ each place you wish to insert the text as part of the output. This can save you typing and conserve the memory space in your Home Computer.

Chapter Summary

In this chapter, you've covered a lot of important ground. You've learned how to:

- Enter a program
- Use the commands NEW, LIST and RUN
- Put messages in your PRINT statements
- Use INPUT statements with *numeric variables* and *string variables*
- Build a mathematical conversion program

When you started working with Chapter Three, you were a beginner in learning BASIC and programming. Now you're well on your way to becoming a computer programmer. Look over the quick review below, and then try your hand at the exercises for this chapter. See you later, Gilgara!

Quick Review of Program Structure

- (1) Begin each line with an identifying line number (1 – 32767).
- (2) Number the lines in the order you want the computer to follow in executing the program.
- (3) Press **ENTER** when you have finished typing a program line.

Chapter Three Exercises

- (1) Programs do not execute immediately. What command is used to execute a program? _____
- (2) What command is used to erase old programs? _____
- (3) What command is used to display the program that is currently stored in memory? _____
- (4) This short program has just been entered.

```

10 CALL CLEAR
30 B=20
40 PRINT B
20 A=10
50 PRINT A

```

In what order will the lines be executed when the program is run?

1 2 3 4 5

- (5) If the program of exercise 4 is in the computer's memory, show what would be displayed after you type:

```

>CALL CLEAR
>LIST

```

- (6) If the program of exercise 5 is run, show what would be on the display.

- (7) If the following line were added to the program example, show the results of the arithmetic operations.

```

60 PRINT A*B; B/A; B+A; B-A

```

A*B = _____ B/A = _____

B+A = _____ B-A = _____

Answers to Chapter Three Exercises

(1) RUN

(2) NEW

(3) LIST

(4) 10, 20, 30, 40, 50

(5)

```
>LIST
10 CALL CLEAR
20 A=10
30 B=20
40 PRINT B
50 PRINT A
>□
```

(6)

```
20
10
**DONE**
>□
```

(7) $A*B = 200$ $A/B = 2$ $B+A = 30$ $B-A = 10$ (8)

```
20 INPUT A
30 INPUT B
```

(9) ?

(10)

```
100 CALL CLEAR ← Optional
200 INPUT "FEET=";F
300 PRINT "INCHES=";F*12 ← Yours may be different – if it works, it's OK
```

(11)

```
DIAMETER OF WHEEL=?20
CIRCUMFERENCE=62.80 INCHES
**DONE**
>□
```

Chapter Four

Looping Sound and Color

All of the programs you have used so far have been executed in a straight-line, sequential order from the smallest line number to the largest line number. After executing the last statement (largest line number), the computer stopped.

For example:

```
1st → 5 CALL CLEAR
2nd → 10 INPUT "KILOGRAMS=?":K
3rd → 20 PRINT 2.2*K;"POUNDS"
4th → 30 END
```

After executing line 30, the computer stopped. There are many situations, however, where you want to interrupt the orderly flow of operation. You may want to repeat a part of a program several times, or you might want to skip over a part of a program based on certain conditions. You can do this by using the GO TO statement.

The GO TO Statement

The GO TO statement is a control statement that causes the program to branch to a specified line number instead of automatically going on to the next higher numbered line. The form of the GO TO statement is:

```
Line number of the
GO TO statement
↓
30 GO TO 10
↑           ↑
Statement  Line number to go to
```

We rewrite our program example, using a GO TO statement in place of the END statement.

```
5 CALL CLEAR
10 INPUT "KILOGRAMS=?":K
20 PRINT 2.2*K;"POUNDS"
30 GO TO 10
      ↙
      GO TO statement
```

Program execution is sequential until the GO TO statement is reached. The GO TO statement interrupts this sequential execution. The line number of the next statement to be executed (statement 10 in this example) is shown in the GO TO statement. Line 10 will be the next program statement executed even if there are other statements following line 30.

Enter the program, then type RUN and press ENTER.

```

    KILLOGRAMS=? ← This is what you see
    
```

Now type 50 as the number of kilograms and press ENTER.

```

    KILLOGRAMS=?50      The computer shows
    110 POUNDS          the answer and then
    KILLOGRAMS=? ← asks again
    
```

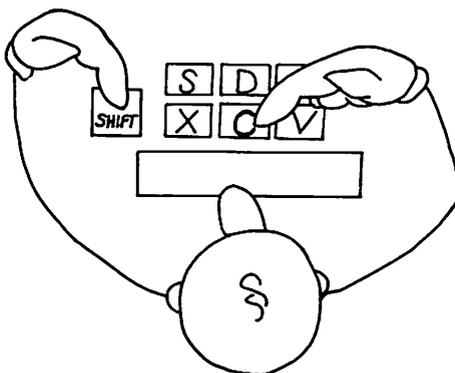
Each time you type the number of kilograms and press ENTER, the computer will print the number of pounds, then ask for kilograms again.

Here is what the screen looked like after we had entered 50, 60 and 70 kilograms.

```

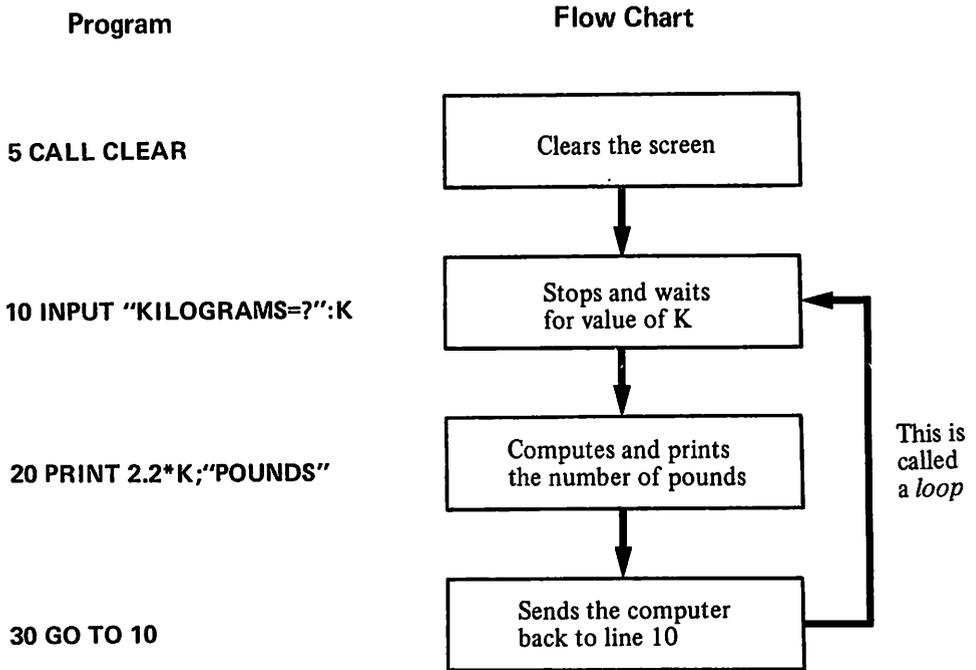
    KILLOGRAMS=?50
    110 POUNDS
    KILLOGRAMS=?60
    132 POUNDS
    KILLOGRAMS=?70
    154 POUNDS
    KILLOGRAMS=?
    
```

How do you stop this program? You press the SHIFT and the C keys together – SHIFT C . This action halts any program execution. Try this command on the current program example.



In executing this program, the computer does line 5, then line 10, then line 20, then line 30, then line 10, then line 20, then line 30, then line 10 and so on. Of course, to do line 10, the computer needs *your* cooperation. You must enter a value of K and press ENTER.

The following diagram, called a *flow chart*, illustrates how the program works.



Experiment. Add the following line to the program.

25 PRINT

RUN the program to see what this does. This “empty” PRINT statement causes a blank line to be printed. The blank line separates the information on the screen for each pass through the loop.

GO TO With CALL SOUND

You can use variables, rather than numbers, in the CALL SOUND statement. For example, let’s use these variables:

D = duration
 F = frequency
 L = loudness

Enter the following program.

```

10 CALL CLEAR
20 INPUT "DURATION=?":D
30 INPUT "FREQUENCY=?":F
40 INPUT "LOUDNESS=?":L
50 CALL SOUND(D,F,L)
60 PRINT
70 GO TO 20
    
```

Caution! Use only legal values for these inputs.
 D: 1 to 4275
 F: 110 to 44,000
 L: 0 to 30

The variables are used in the CALL SOUND statement

Now RUN the program. Enter values for duration, frequency and loudness. After you enter the value for loudness and press **ENTER**, the computer will play your tone, then ask for new values. Here is a sample RUN.

```

DURATION=?1000
FREQUENCY=?262
LOUDNESS=?10
          

DURATION=?4275
FREQUENCY=?440
LOUDNESS=?23
          

DURATION=?
    
```

Play as many tones on your computer as you wish. When you want to stop, press **SHIFT C**.

Note the spacing in the above RUN. Each set of values (duration, frequency, loudness) is separated from the previous set by a vertical space. This empty space is courtesy of line 60.

60 PRINT Tells the computer to print an "empty" line.

Delete line 60 and RUN the program again. Without line 60, no spacing will occur between sets of values. Which do you prefer?

```

DURATION=?1000
FREQUENCY=?262
LOUDNESS=?10
DURATION=?4275
FREQUENCY=?440
LOUDNESS=?23
DURATION=?
    
```

Loops

Study the following program, but don't enter it yet.

```

10 CALL CLEAR
20 INPUT K
30 PRINT K
40 PRINT
50 K=K+1
60 GO TO 30

```

} A loop!

Here we "send" the program back to line 30 by using a GO TO statement in line 60. The GO TO statement causes the actions performed by lines 30, 40, and 50 to be repeated over and over again, setting up a *loop*. (Notice that we don't need to use an END statement.) It won't stop until you tell it to by pressing **SHIFT C**. This example sets up an "endless loop."

Let's enter the program now. First, type **NEW** and press **ENTER** to erase the computer's program memory, and then type these lines:

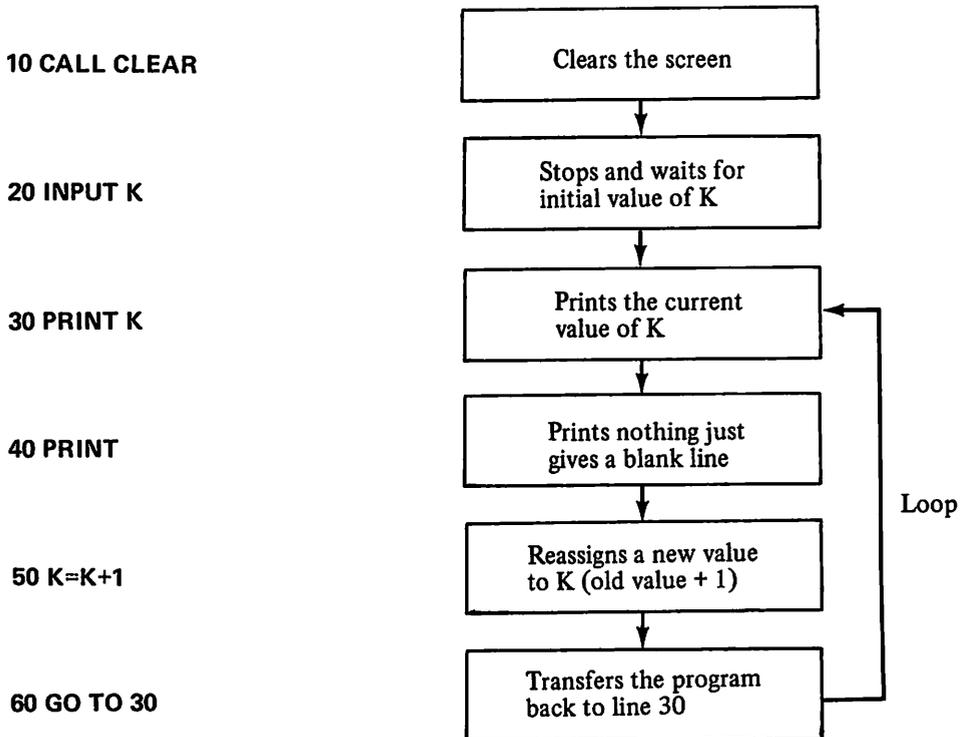
```

10 CALL CLEAR
20 INPUT K
30 PRINT K
40 PRINT
50 K=K+1
60 GO TO 30

```

← Note: the LET is optional

Before you run the program, let's look at the flow chart, showing how the program works.



If you trace the actions of the program step by step and look at the results, this “looping” can be easily visualized.

Statement	Trace K	Remarks
10 CALL CLEAR	0	K stands at zero
20 INPUT K	1	You input a 1 for K
30 PRINT K	1	1 is displayed on the screen
40 PRINT	1	Skip a line on the screen
50 K=K+1	2	K gets a new value, 1+1=2
60 GO TO 30	2	Directions to go back to line 30
30 PRINT K	2	2 is displayed on the screen
40 PRINT	2	Skip a line on the screen
50 K=K+1	3	K gets a new value, 2+1=3
60 GO TO 30	3	Go back to line 30 again
30 . . .		Will the program end?

Now run the program, putting in 1 for the beginning value of K. Watch how quickly the computer counts – almost too fast to follow! That’s why we added the blank line (line 40), which spaces out the numbers a bit so that you can see them better.

Let the computer count as long as you want. When you are ready to stop the program, hold down the **SHIFT** key and press **C**. You’ll see “BREAK AT LINE (number)” on the screen, indicating where the program stopped. Run the program several times, using different numbers for the initial value of K (50, 1000, 5000, etc.).

GO TO can also be typed as GOTO in your program. The computer isn’t fussy about that, so long as the line number contained in the statement (30 in the example) is actually in your program.

Two important requirements exist when using the GO TO statement.

- (1) A valid line number must follow the words GO TO (or the word GOTO).
- (2) The specified line number must be in the program. (Don’t reference a line that is not there.)

If you try to send the program to a nonexistent line number, you’ll get an error message.

Suppose, for example, we type in

```
60 GO TO 25
```

and press **ENTER**. Try it, run the program, and see what happens!

```
10 CALL CLEAR
20 INPUT K
30 PRINT K
40 PRINT
50 K=K+1
60 GO TO 25
```

```
? 1
1
* BAD LINE NUMBER IN 60
>□
```

Now, correct the line

```
60 GO TO 30
```

and run the program again. The computer now merrily counts away until you stop it.

The computer is not limited to counting by ones. You can make it count by 2's, 3's, 4's, or whatever by changing line 50. Let's make the computer count by 2's. Type:

```
50 K=K+2
```

and press **ENTER**. Now run the program, typing in 1 when the computer asks for the starting value of K. The computer counts 1, 3, 5, 7, 9, 11, 13, etc. You probably recognize these as the positive odd integers.

You do not have to start K at 1 every time. Run the program again. When the computer asks for the starting value of K, type in 2. Now the computer gives you the positive even integers: 2, 4, 6, etc.

Can we make the computer count backwards? Of course we can. Type:

```
50 K=K -1 and press ENTER
```

Now, when the computer asks for a starting value, input 25 for K. It will quickly count: 25, 24, 23, 22, etc. After 0, the negative integers will appear: -1, -2, -3, -4, etc. until you press **SHIFT C** to stop the run.

Experiment with the program for a while, making it count by 3's, 5's, 10's, etc. Also try various values for the starting number. Try some negative values. Make it count backwards as well as forward. Since you now know the computer can count, let's move on to other interesting examples with the **GO TO** statement.



Musical Scales

GO TO loops have many applications, of course, beyond simple counting. We could use a loop, for example, to play a musical scale.

DEMO GO TO PROGRAM NO. 1

```

10 LET DURATION=100 }
20 LET TONE=110      } ← Original sound values
30 LET LOUDNESS=2   }
40 CALL SOUND(DURATION,TONE,LOUDNESS) ← This makes the sound
50 TONE=TONE+15     ← Increase the tone
60 GO TO 40         ← Go back and make a new sound

```

Before running the program, you should remember that the program will be running in an endless loop. It will terminate in one of two ways:

- (1) You may terminate it at any time by pressing **SHIFT C** on the keyboard.
- (2) If the tone value goes out of range (above 44,000), automatic termination will occur.

Run the program and listen to the tones that are made. If your ear objects to the sounds created by Program 1, you will appreciate Program 2. Since the notes of the normal musical scale are not exactly 15 units apart, Program 1 may produce unpleasant sounds. Let's try other values for TONE that provide a one-octave scale.

DEMO GO TO PROGRAM NO. 2

```

10 LET T=100 ← "T" for "time"
20 LET V=2   ← "V" for "volume"
30 C=262 ← frequency of middle C on the piano.
40 D=294 ← Note that the word LET is optional
50 E=330
60 F=349
70 G=392
80 A=440
90 B=494
100 HIC=523 ← "HIC" for "high C"

```

This is a C scale

Now you're ready for the CALL SOUND statements to tell the computer when to play each note:

```

200 CALL SOUND(T,C,V)
300 CALL SOUND(T,D,V)
400 CALL SOUND(T,E,V)
500 CALL SOUND(T,F,V)
600 CALL SOUND(T,G,V)
700 CALL SOUND(T,A,V)
800 CALL SOUND(T,B,V)
900 CALL SOUND(T,HIC,V)

```

Finally, create a loop with a GO TO statement:

```
950 GO TO 200
```

Check the program for errors, and correct any that you find. When everything is correct, run the program. Again, this is an endless loop. You'll have to press **SHIFT C** to stop it.

Up, up, up you go until you reach high C. Then the GO TO statement at line 950 sends you back to middle C to start over.

STOP THE PROGRAM! Now reverse the order of lines 200–900.

```
200 CALL SOUND(T,HIC,V)
300 CALL SOUND(T,B,C)
400 CALL SOUND(T,A,V)
500 CALL SOUND(T,G,V)
600 CALL SOUND(T,F,V)
700 CALL SOUND(T,E,V)
800 CALL SOUND(T,D,V)
900 CALL SOUND(T,C,V)
```

After you have entered the program changes, run the program. Down, down, down it goes from high C to middle C. Once again, line 950 causes the sounds to be repeated over and over. Press **SHIFT C** when you tire of the program. This is much easier than practicing the scales on the piano.

You have played an octave both up and down. Now let's put them together so that you go Up, Down, Up, Down, etc. Type NEW and enter this program:

DEMO GO TO PROGRAM NO. 3

```
10 LET T=100
20 LET L=2
30 C=262
40 D=294
50 E=330
60 F=349
70 G=392
80 A=440
90 B =494
100 HIC=523
200 CALL SOUND(T,C,L)
210 CALL SOUND(T,D,L)
220 CALL SOUND(T,E,L)
230 CALL SOUND(T,F,L)
240 CALL SOUND(T,G,L)
250 CALL SOUND(T,A,L)
260 CALL SOUND(T,B,L)
270 CALL SOUND(T,HIC,L)
280 CALL SOUND(T,B,L)
290 CALL SOUND(T,A,L)
300 CALL SOUND(T,G,L)
310 CALL SOUND(T,F,L)
320 CALL SOUND(T,E,L)
330 CALL SOUND(T,D,L)
340 GO TO 200
```

← T for Time
← L for Loudness

} Assigned values for notes of one octave

} Each note is called separately

← Go back and repeat it all

A GO TO Loop with the CALL COLOR Statement

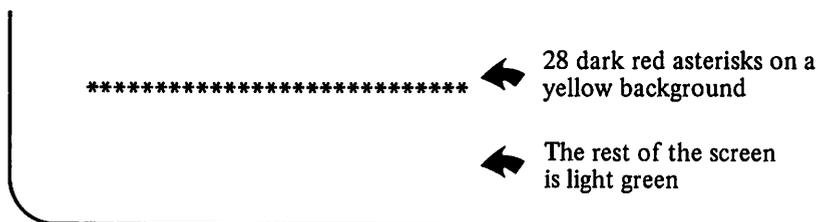
Up to now, you've seen only three colors in BASIC on your Home Computer (maybe you've only noticed two, but there really are three). First, while you're entering a program the screen background is light blue, and the characters (letters and numbers) that you're typing are black. Then, while the program is running, the screen becomes a light green color. When the program stops, the screen returns to light blue with black characters.

These are only three of the sixteen colors available with your computer. The way you control all the colors with a program is through the CALL COLOR statement. Let's try a program with a CALL COLOR statement and a slightly different GO TO loop. Clear your old program from the computer's memory (NEW, press ENTER), and type these lines:

```
'10 CALL CLEAR
20 CALL COLOR(2,7,12)
30 CALL HCHAR(12,3,42,28)
40 GO TO 40 ← A GO TO loop that "goes to" itself!
```

Line 40 shows another use of the GO TO statement. Instead of going back to a previous line to repeat a portion of the program, it merely repeats itself. It's just like idling a motor. The computer is doing nothing but going round and round on line 40. The purpose of this is to prevent the screen from scrolling the display up one line. Ordinarily, when a program comes to the end, the display is scrolled up to print ****DONE**** and up again to display the prompt and cursor. Line 40 prevents this from happening, and the characters on the display will stay exactly where we put them.

Now run the program, and the screen should look like this:



Our program prints twenty-eight asterisks across the screen. The asterisks are dark red, and in the area where they are displayed, the screen color is a light yellow. The rest of the screen remains light green.

Remember, line 40 puts your program into a kind of "holding pattern" that keeps your graphic on the screen.

When you're ready to stop the program, hold down the **SHIFT** key and press **C** to "break" the loop. You can run the program as many times as you like.

A CALL COLOR statement requires three numbers, enclosed in parentheses and separated by commas:

20 CALL COLOR(2,7,12)

The first number after the open parenthesis symbol is a *character set number*. As we mentioned in Chapter One, each *character* (letters, numbers, and symbols) that prints on the screen has its own numeric code, ranging from 32 through 95 for a total of 64 characters. These characters are organized by the computer into eight *sets* with eight characters each:

Set No. 1		Set No. 2		Set No. 3		Set No. 4	
Code No.	Character						
32	(space)	40	(48	0	56	8
33	!	41)	49	1	57	9
34	"	42	*	50	2	58	:
35	#	43	+	51	3	59	;
36	\$	44	,	52	4	60	<
37	%	45	-	53	5	61	=
38	&	46	.	54	6	62	>
39	'	47	/	55	7	63	?

Set No. 5		Set No. 6		Set No. 7		Set No. 8	
Code No.	Character						
64	@	72	H	80	P	88	X
65	A	73	I	81	Q	89	Y
66	B	74	J	82	R	90	Z
67	C	75	K	83	S	91	[
68	D	76	L	84	T	92	/
69	E	77	M	85	U	93]
70	F	78	N	86	V	94	^
71	G	79	O	87	W	95	-

(As a handy reference, these character sets and code numbers are also listed in the Appendix.)

The set number you use in a CALL COLOR statement, then, is determined by the character you want to print. (And what happens if you want to print characters from different sets in the same colors? We'll discuss that shortly.)

20 CALL COLOR(2,7,12)

character set no. 2



The second and third numbers determine the colors used in your graphic. Each of the sixteen colors has its own numeric code.

Color	Code No.	Color	Code No.
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

The second number sets the *foreground* color, that is, the color of the character you designate. The third number sets the *background* color – the color of the block or square in which the character is printed.

20 CALL COLOR(2,7,12)

foreground color – dark red background color – light yellow

The next line in your program is

30 CALL HCHAR(12,3,42,28)

Row number Column number Numeric code for * Number of repetitions

(If you need to review the CALL HCHAR examples in Chapter One, this would be a good time to do it.)

Now you know why we indicated Set No. 2 in our CALL COLOR statement. The asterisk (code number 42) is a part of Set No. 2.

Line 40 of the program is a GO TO statement that “goes to” itself. It keeps the computer “idling” until you press **SHIFT C**.

When you run the program, the screen looks like this. Note the colors!

```

*****

```

← 28 dark red asterisks on a yellow background

← The rest of the screen is light green

But when you press **SHIFT C**, the program stops, and the screen changes back to its normal blue color. All the reds, yellows, and greens disappear. Line 40 not only keeps the screen from scrolling, but it also keeps the desired colors on the screen for you to see.

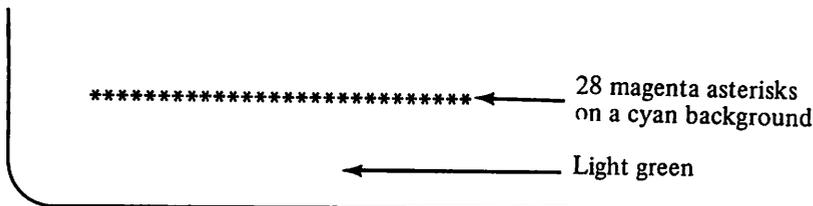
Now let's change line 20 of the program to see some new colors. Stop the program, if it's still running, and type this:

```
20 CALL COLOR(2,14,8)
```

Press **ENTER** to store your new line, and list the program (**LIST**; press **ENTER**) to review your program.

```
LIST
10 CALL CLEAR
20 CALL COLOR (2,14,8)
30 CALL HCHAR(12,3,42,28)
40 GO TO 40
```

When you're ready, run the program.



You could, of course, continue to experiment by stopping the program, entering a new line 20, and running the modified program over and over. Don't. Instead, save wear and tear on your fingers by entering the following program which allows you to experiment more easily. With this program, you enter foreground (F) and background (B) colors in response to **INPUT** statements.

```
NEW
10 CALL CLEAR
20 INPUT "FOREGROUND ":F } ← INPUT color codes
30 INPUT "BACKGROUND ":B }
40 CALL COLOR(2,F,B) ← Set colors based on INPUT color codes
50 CALL HCHAR(12,3,42,28)
60 GO TO 60 ← You know what this does
```

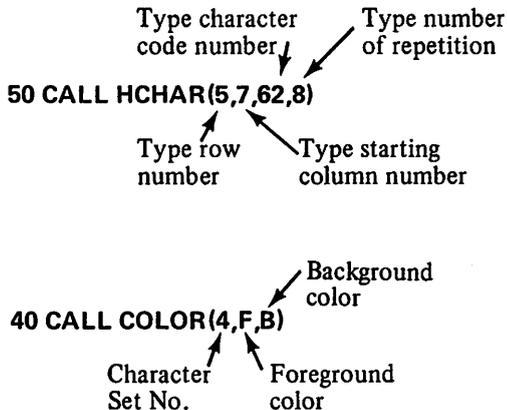
When the computer asks you for the "foreground" and "background" colors you want to use, you can type in any color code from 1 through 16. Remember, however, that color number 1 is "transparent." This "color" has important applications that are discussed later in the book, but are not satisfactory in the current program. Also, color number 2, black, can cause display distortion on some color monitors. You may want to use only the colors 3 through 16 in running this program. Here are some combinations you might find interesting:

Foreground Color	Background Color
3	16
3	11
5	6
5	14
7	15
7	12
13	12
14	10

O.K., have you checked your program for typographical errors? Have you chosen the foreground and background colors you want to use first? Then run the program.

Experiment with other color combinations so that you will know which look good and especially those that do not go well together. You might want to write down your findings for future reference.

After you've experimented with different color combinations, you might enjoy trying some other characters. You can do this by retyping line 50, substituting a different character code number for the "42" asterisk code number. Just remember, if you select a character from any set other than No. 2, you'll also have to change line 40 to reflect the new set number. For example:



The change to line 50 causes a right pointing caret (>) to be displayed. The caret, whose character code is 62, is from character set no. 4. To alter the color of this character, line 40 must also be changed. Run the program with these changes while trying various color combinations. Then make some character changes on your own.

What if you want to print characters from different sets, all in the same color? One way to do this is to include in your program eight `CALL COLOR` statements, one for each of the eight sets of characters. You'll have to do quite a bit of typing, but you'll be free to use any of the characters you choose. You *do not have to use* every character set; but if you want to, you can. The eight `CALL COLOR` statements allow you to use any character from code number 32 through 95.

Try this program:

```

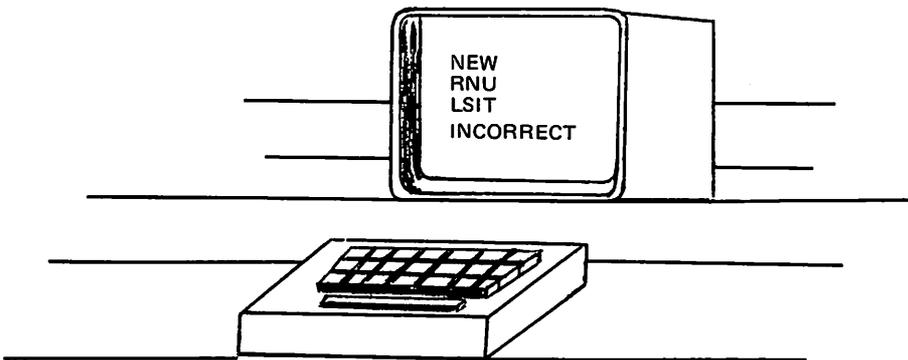
NEW
100 CALL CLEAR
110 CALL COLOR(1,6,16) ← Light blue
120 CALL COLOR(2,6,16) ← White
130 CALL COLOR(3,6,16)
140 CALL COLOR(4,6,16)
150 CALL COLOR(5,6,16)
160 CALL COLOR(6,6,16)
170 CALL COLOR(7,6,16)
180 CALL COLOR(8,6,16)
190 PRINT " " ← You decide what to do here!
200 GO TO 200
    
```

Use any message you want in line 190; just remember to enclose it in quotation marks. With these CALL COLOR statements you have told the computer to print all of the sixty four characters in light blue (6) on a white (16) background.

There are other program examples with CALL COLOR later in the book. For now, experiment on your own with various colors and character sets. Put a little COLOR in your life!

Error Messages

We haven't talked much in this chapter about error messages because, for the most part, the ones you'd run into in these program examples are the same as – or even similar to – those you learned about in Chapter One. For example, a spelling or typing error in NEW, RUN, or LIST will cause the computer to return an "INCORRECT STATEMENT" message as soon as you press ENTER.



Errors in program statements may be detected by the computer either when the line is entered *or* when the program is run. Here are some samples of error conditions and messages you might see:

Condition	Message
Omitting a quotation mark: 10 INPUT "WHAT COLOR:F	*INCORRECT STATEMENT
Misspelling a statement: 10 INPU "WHAT COLOR":F	*INCORRECT STATEMENT IN LINE 10
Omitting necessary punctuation or typing an incorrect punctuation mark: 10 INPUT "WHAT COLOR" F 10 INPUT "WHAT COLOR";F	
Leaving the variable out of an INPUT statement: 10 INPUT "WHAT COLOR":	
Leaving out the space between GO TO and the line number: 10 GO TO30	
Using a nonexistent line number in a GO TO statement: 10 LET A=5 20 PRINT A 30 GO TO 15	*BAD LINE NUMBER IN LINE 30 ← There is no line 15!

Notice that the error messages given during a program run indicate the number of the troublesome line. If you'd like to view the line in question (let's say it's line 10), just type

LIST 10

and press **ENTER**. The computer will obediently print line 10 on the screen for you to review.

You can also list the whole program on the screen if you prefer. Type

LIST

and press **ENTER**.

Remember, too, that failing to press **ENTER** at the end of each program line may cause the computer to give you an error message or an incorrect result, depending on the kind of operation you're performing.

Most of the possible errors in the use of the SOUND and COLOR routines have to do with using out-of-range values as parameters (e.g., a tone parameter that is too large, an incorrect color specification, a character number that is not allowed).

Making mistakes is a normal part of learning – so don't be disturbed when the computer gives you an error message. Just list the line or the program, identify the error, retype the line correctly, and go right on your way.

(*Note:* If you'd like to see all the error messages your computer can give you, or if you don't understand a message you're given, you'll find a complete list of error messages – and when they occur – in the “BASIC Reference Section” of your *User's Reference Guide*.)

Chapter Summary

In this chapter you've covered a lot of very important ground. You've learned how to:

- Create a GO TO loop within a program.
- Stop an endless loop with **SHIFT C**
- Trace the steps of a program as the computer would perform them
- Use a GO TO loop in a CALL SOUND program
- Play a one-octave scale using CALL SOUND
- Use the CALL COLOR statement with a GO TO loop to create color graphics
- Select colors for both foreground and background display
- Experiment with display colors of your choice

You're well on your way to becoming an experienced computer programmer.

Chapter Four Exercises

- (1) Study this program.

```

10 CALL CLEAR
20 INPUT K
30 PRINT 2.2*K
40 GO TO 20
50 END

```

- (a) Write the order in which the computer will execute lines for the first eight steps. (1) _____ (2) _____ (3) _____ (4) _____ (5) _____ (6) _____ (7) _____ (8) _____
- (b) Will line 50 ever be executed? _____
- (2) Write the results displayed on the screen for the first four times the computer passed through the GO TO loop of this program.

```

100 CALL CLEAR
110 K=9
120 PRINT K
130 K=K*2
140 GO TO 120

```

- (a) 1st time _____
- (b) 2nd time _____
- (c) 3rd time _____
- (d) 4th time _____

- (3) Write the first four results if line 140 of the program in exercise 2 was changed to:

```

140 GO TO 110

```

- (a) 1st time _____
- (b) 2nd time _____
- (c) 3rd time _____
- (d) 4th time _____
- (4) What would happen if line 140 of the same program was changed to:

```

140 GO TO 125

```

- (5) Rewrite the program of exercise 2 so that the first K will be 81 and each loop will *divide* the previous number by 3.

```

100 _____
110 _____
120 _____
130 _____
140 _____
    
```

Notice the strange stuff appearing on the screen when you run this program.

- (6) How do you stop a program that is in an endless GO TO loop? _____

- (7) Answer the questions below for this program.

```

100 CALL SOUND(200,262,2)
110 CALL SOUND(100,262,2)
120 CALL SOUND(200,294,2)
130 CALL SOUND(200,294,3)
140 GO TO 100
    
```

- (a) Describe the difference between the sounds played by lines 100 and 110.

- (b) Describe the difference between the sounds played by lines 100 and 120.

- (c) Describe the difference between the sounds played by lines 120 and 130.

- (8) Write a program that will let you input the frequency of the note so that you can play your own music. Use 500 for the time duration and 2 for volume. Be sure to use a GO TO statement so that your piece of music can be as long as you want.

```

_____
_____
_____
_____
    
```

- (9) (a) What is the background color of the screen when you are entering a program?

- (b) What color is the background of the screen when a program is running?

(10) Tell what each number in this CALL COLOR statement represents.

CALL COLOR(2,15,11)

2 _____

15 _____

11 _____

(11) The up-arrow (↑) is in character set no. 8. Its code number is 94. The color code for black is 2 and for white is 16. Complete this program to put 10 horizontal, black up-arrows on a white background starting in row 3, column 5. Clear the screen first and use a GO TO loop to hold the characters on the screen.

100 _____

110 _____

120 _____

130 _____

(12) Rewrite the program above so that you can input the foreground and background colors. Use F for the foreground variable and B for background.

80 _____

90 _____

100 _____

110 _____

120 _____

130 _____

Answers to Chapter Four Exercises

(1) (a) 10, 20, 30, 40, 20, 30, 40, 20

(b) No, line 40 always sends the computer back to line 20

(2) (a) 1st time 9

(b) 2nd time 18

(c) 3rd time 36

(d) 4th time 72

(3) (a), (b), (c), (d) all 9 (line 110 would always reassign 9 to K before it was printed.)

(4) An error message would be printed, and the program would stop.

(5) **100 CALL CLEAR**
100 K=81
120 PRINT K
130 K=K/3
140 GO TO 120

(6) Press **SHIFT C**

- (7) (a) The note of line 100 lasts twice as long as that of line 110.
 (b) Different frequencies, but same length. (Line 100 is middle C, line 120 is D above middle C.)
 (c) Volume changed (line 120 is louder).

(8) Our program (yours may be different – try it to see if it works):

100 T=500	or	100 INPUT N
110 V=2		110 CALL SOUND(500,N,2)
120 INPUT N		120 GO TO 100
130 CALL SOUND(T,N,V)		
140 GO TO 120		

- (9) (a) light blue
 (b) light green

(10) 2 is the character set number
 15 is the foreground color (gray)
 11 is the background color (dark yellow)

(11) **100 CALL CLEAR**
110 CALL COLOR(8,2,16)
120 CALL HCHAR(3,5,94,10)
130 GO TO 130

(12) **80 INPUT "FOREGROUND ":F**
90 INPUT "BACKGROUND ":B
100 CALL CLEAR
110 CALL COLOR(8,F,B)
120 CALL HCHAR(3,5,94,10)
130 GO TO 130

Chapter Five

More Programming Power

By now you've done quite a bit of programming in TI BASIC. You know what a program is, how it's built, and what happens when you run it through a computer. Congratulations! You are now ready to add several new BASIC statements to your toolbox of programming skills.

First, there is the useful and versatile FOR-NEXT statement. This statement is used to create loops in programs – not “endless” loops but loops that “know” when to stop. With FOR-NEXT loops, you can direct your program to “wait for a period of time” – to delay operations. Delay loops can “slow down” parts of your program thus slowing down the action on the screen.

However, the FOR-NEXT statement does more than just put delays in programs. You will discover that the clever use of FOR-NEXT loops can let you do amazing things with your Home Computer – from arithmetic to graphic designs. The FOR-NEXT statement is one of the more powerful BASIC language features of your computer.

The second set of BASIC statements you will see in this chapter are GOSUB and RETURN. If you are a beginning typist, you will like what these two statements can do for you. GOSUB and RETURN can save you a lot of typing. They allow you to reuse a section of your program without having to type it in again.

All of these new language features will help increase your programming skills. You will build on what you've learned in previous chapters and prepare yourself for the many exciting things yet to come. So let's begin with FOR-NEXT loops!

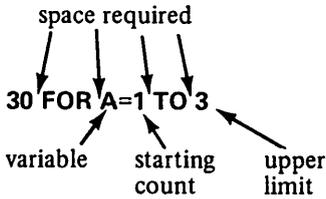
The FOR-NEXT Statement

Chapter Four presented several examples of the GO TO loop. Each example repeated a statement or a set of statements indefinitely – until you pressed **SHIFT C**, and stopped the program. The FOR-NEXT statement also creates a loop, but it's different from GO TO in two ways:

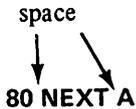
- (1) The FOR-NEXT statement is actually a pair of lines in the program, the FOR line and the NEXT line, each with its own line number. The GO TO statement is always on one line.

- (2) You control the number of times the FOR-NEXT loop operates. After the loop runs the number of times you specify, the program moves on to the line following the NEXT line. The GO TO statement continues looping until you press **SHIFT C**.

The beginning line (or FOR line) has the form:



The ending line (or NEXT line) might appear as follows:



the same variable used in the FOR line

How would you build a program with a FOR-NEXT loop? It is really quite simple. Erase any current programs by typing **NEW** and pressing **ENTER**. Now enter the following lines and **RUN** the program.

```

10 CALL CLEAR
20 FOR A=1 TO 3
30 PRINT "A=";A
40 NEXT A
50 PRINT "OUT OF LOOP"
60 PRINT "A=";A
70 END
    
```

← FOR-NEXT loop

On the screen you see:

```

A= 1      Once around the loop
A= 2      Twice around the loop
A= 3      Thrice around the loop
OUT OF LOOP  Now we are out because
A= 4      A is over the limit
**DONE**
>□       What's next?
    
```

How did it do that? Here is a line-by-line description showing the order in which the lines were executed. See if you can follow the computer's action in this *trace* of the program.

Line	Statement		
Line	Statement	Value of A After Statement Execution	Remarks
10	CALL CLEAR	0	Screen is cleared
20	FOR A=1 TO 3	1	Loop begins. A is set to 1 and the upper limit is set to 3
30	PRINT "A=";A	1	A=1 is printed.
40	NEXT A	2	A is increased by 1 and checked. against the upper limit. Still within limit; branch to 30
30	PRINT "A=";A	2	A=2 is printed.
40	NEXT A	3	A is increased again; checked against upper limit. Still within limit; branch to 30.
30	PRINT "A=";A	3	A=3 is printed.
40	NEXT A	4	A is increased; now beyond upper limit.
50	PRINT "OUT OF LOOP"	4	Display out of loop.
60	PRINT "A=";A	4	A=4 is printed.
70	END		Program stops!

You can change the program by changing the values in the FOR statement. Try the following changes (one at a time, of course).

(1) 20 FOR A=4 TO 6

```

A= 4
A= 5
A= 6
OUT OF LOOP
A= 7
**DONE**
>□

```

(2) 20 FOR A=3 TO 9

(3) 20 FOR A=11 TO 20

Are you getting a feel for how the FOR-NEXT loop works? If yes, GOOD! You are ready to move on. If not, try some more changes to line 20. Then read on as we show some more examples. Pick a large value for the upper limit that causes the printed values to scroll off the screen. What happens if you give A an initial value of 1/10 (or .1) and an upper limit of 1/2 (or .5)? Try it and see what appears on the screen.

The FOR-NEXT Delay Loop

In Chapter Four, the GO TO statement was used in some of the CALL COLOR programs in this way:

```
40 GO TO 40
```

This line created a delay or “idling” loop in the program. The color design on the screen was held in place by this kind of loop until you pressed **SHIFT C** to stop the program. Without the “idle” loop, the color design would just blink on the screen and then disappear. You would see a momentary “flash” and the screen would return to the standard Immediate Mode color patterns. The delay loop gave you a chance to see the program’s color design.

The FOR-NEXT statement can be used to create delay loops in your programs. The major difference is the FOR-NEXT loops allow you to build a controlled time delay — one that “idles” for a time and then continues with the program execution. This feature gives you great flexibility in designing and printing information on the screen.

Lines 50–60 in the next example use the FOR-NEXT statement as a time delay. The computer makes 200 trips through this loop before going to line 70. Although no other statement is executed within the loop, the FOR-NEXT allows time for you to view the color bar that is displayed.

COLOR BAR PROGRAM 1

10 CALL CLEAR		Clear screen
20 INPUT A		Input a color number (1–16)
30 CALL COLOR(2,A,A)		Set color
40 CALL HCHAR(12,3,42,28)		Display bar
50 FOR B=1 TO 200	←	Delay loop
60 NEXT B		
70 GO TO 10		Go back for new input

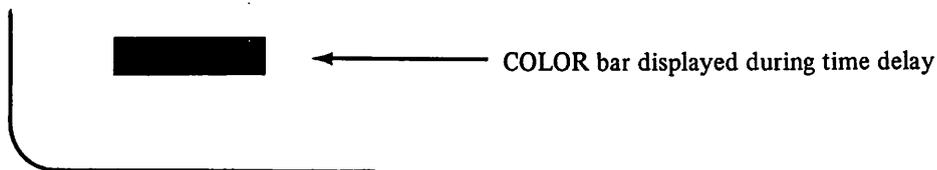
The program requires that you enter a number between 1 and 16. Each of these numbers selects a different color for the display. (You will find a list of the color codes available on your Home Computer in the Appendix.) After the time delay, the GO TO statement sends the program back for a new input.

In line 40 of the program, the CALL HCHAR places 28 characters on row 12 of the screen, starting at column 3.

```
40 CALL HCHAR(12,3,42,28)
```

Notice that although character 42 (the * symbol) is used, only a bar of color appears on the screen. Since the foreground and background colors are the same, the character “disappears.” The operation is like writing on blue paper with blue ink, or on white paper with white ink. You may try other character numbers from character set 2 in place of the 42 to verify that all do vanish, leaving only a bar of color.

Now run the program. Does the color bar stay on the screen long enough for you to observe it carefully? If not, change line 50 to increase the time delay (1 TO 2000 for example).



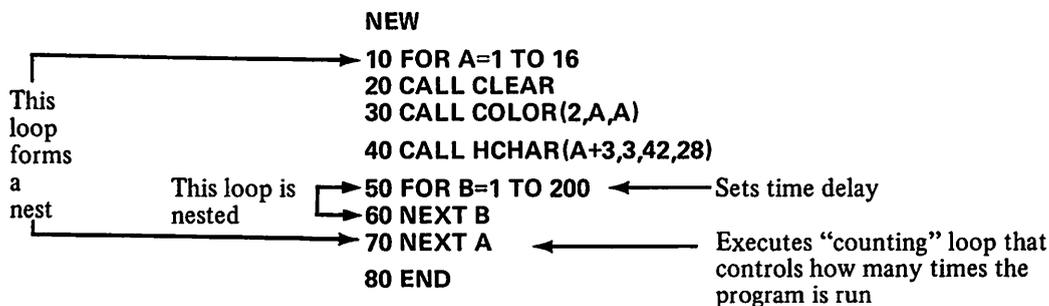
When you stop the program, what happens? Yes, the color bar disappears and you are left with a row of asterisks!

“Nested” FOR-NEXT Loops

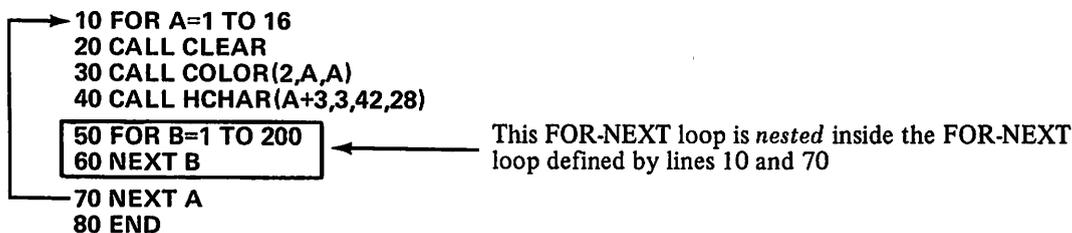
You’ve just seen that we can use both a FOR-NEXT loop and a GO TO loop in the same program. It’s also possible for you to use more than one FOR-NEXT loop – one inside another – in a program. We call these “nested” loops.

As an example, let’s experiment with a program very similar to the one you’ve just completed, but this time we’ll get a little fancier. We’ll make the bar “walk” down the screen, so that it appears in a different position each time the color changes.

Type these lines:

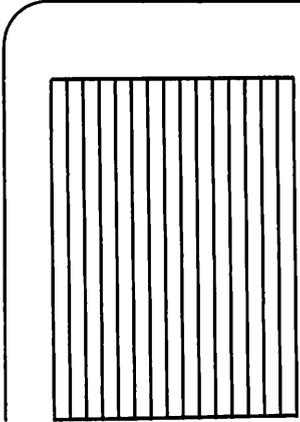


Another way to look at the nest.



When you run the program this time, the bar will be vertical and will move across the screen from left to right (columns 6 through 21).

The display:



Subroutines

In the color bar program, we used a time delay loop nested in another loop (lines 50 and 60). Quite often, in longer programs, a time delay is needed several times. Rather than write the time delay every time it's needed, you could place it outside your main program in a *subroutine*. When you want the time delay, you merely write the statement:

`GOSUB XX` (where XX is the first line number of the subroutine)

After the time delay has been completed the statement

`RETURN`

(which is the last line of the subroutine) sends the program back to the main program.

We'll use the color bar program as an example.

MAIN PROGRAM

```

10 FOR A=1 TO 16
20 CALL CLEAR
30 CALL COLOR(2,A,A)
40 CALL HCHAR(A+3,3,42,28)
50 GOSUB 100
60 NEXT A
70 END

```

SUBROUTINE

```

100 FOR B=1 TO 200
110 NEXT B
120 RETURN

```

The subroutine is "called"

Line 120 returns to the main program at the line following the place it left

The same subroutine may be used many times in the same program. When a GOSUB is executed, the computer “remembers” the line number *where it left* the main program. It always comes back to the *next* line of the main program.

Let’s enlarge the color bar program so that it first displays walking horizontal bars, then follows with vertical bars.

```

COLOR BAR TWO
Loop 1
10 FOR A=1 TO 16
20 CALL CLEAR
30 CALL COLOR(2,A,A)
40 CALL HCHAR(A+3,3,42,28)
50 GOSUB 200
60 NEXT A

Loop 2
70 FOR A=1 TO 16
80 CALL CLEAR
90 CALL COLOR(2,A,A)
100 CALL VCHAR(A,A+5,42,24)
110 GOSUB 200
120 NEXT A
130 END

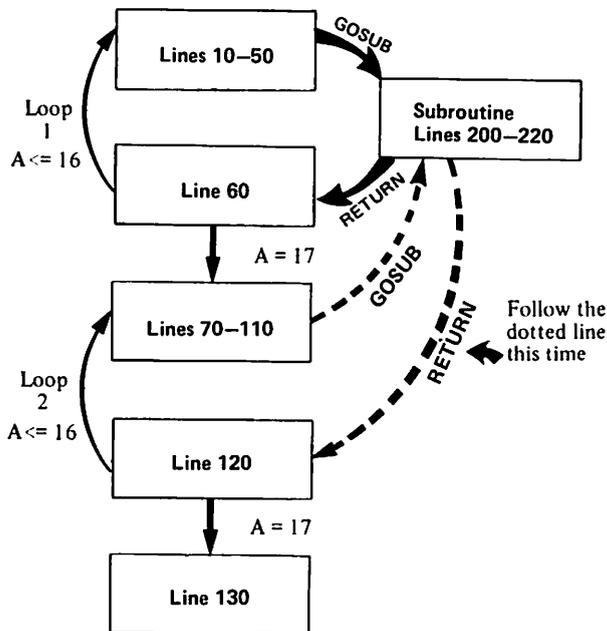
200 FOR B=1 TO 200
210 NEXT B
220 RETURN
    
```

In this part of the program the subroutine returns to line 60 each time through the loop

In this part of the program the subroutine returns to line 120 each time through this loop

← Go back to where you came from plus one line

The flow of program execution may be seen in the next diagram. At line 50 the program goes to subroutine 200 each time the program makes a pass through loop 1, returning to line 50 each time. At line 110, the program goes to subroutine 200 each time the program makes a pass through loop 2, returning to line 120 each time.



Subroutines may be called over and over again in the same program, and each time it will return to the correct spot in the main program. The one condition that is necessary for this to happen is that the word (or statement) RETURN must be in the subroutine, usually the last line of it.

Now let's examine another program with nested FOR-NEXT loops. The following program displays all 64 of the alphanumeric characters, codes 32 through 95. (See the Appendix for a list of the character codes.) Enter these lines:

NEW		
10 CALL CLEAR		
20 LET CHAR=32	←	Starting value for variable CHAR (character code)
30 FOR ROW=7 TO 14	←	Beginning and ending values for row number
40 FOR COLUMN=13 TO 20	←	Beginning and ending values for column number
50 CALL HCHAR(ROW,COLUMN,CHAR)		
60 CHAR=CHAR+1	←	Increases numeric code for CHAR by 1
70 NEXT COLUMN		
80 GOSUB 200		
90 NEXT ROW		
100 END		

The program will look like this on the screen:

```

TI BASIC READY
>10 CALL CLEAR
>20 LET CHAR=32
>30 FOR ROW=7 TO 14
>40 FOR COLUMN=13 TO 20
>50 CALL HCHAR(ROW, COLUMN, CHAR)
>60 CHAR=CHAR+1
>70 NEXT COLUMN
>80 GOSUB 200
>90 NEXT ROW
>100 END
>□

```

There are several things we'd like to point out about this program. First, FOR-NEXT loops *do not* have to start counting at 1. They can begin with whatever numeric value you need to use. Second, the nested loop (FOR COLUMN-NEXT COLUMN) is not just a time delay loop. It actually controls a part of the program repetition. Third, a time delay *is* called after each row of characters is printed (line 80).

Finally, line 50 is called a "wrap-around" line. Since it has more than 28 characters, part of it prints on another line on your screen. This is an important point: *Program lines can be more than one screen-line long.* In fact, a program line, in general, can be up to four screen lines (112 characters) in length. (The exception is the DATA statement. See the "BASIC Reference" section of the *User's Reference Guide* for an explanation.) Notice that a wrap-around line is *not* preceded by the small "prompting" symbol.

The subroutine, which must also be entered, is familiar to you by now. In this program, it imparts a pleasing rhythm to the output of the displayed characters.

```
200 FOR B=1 TO 100
210 NEXT B
220 RETURN
```

Run the program, and the 64 characters will be printed in nice, neat rows on the screen:

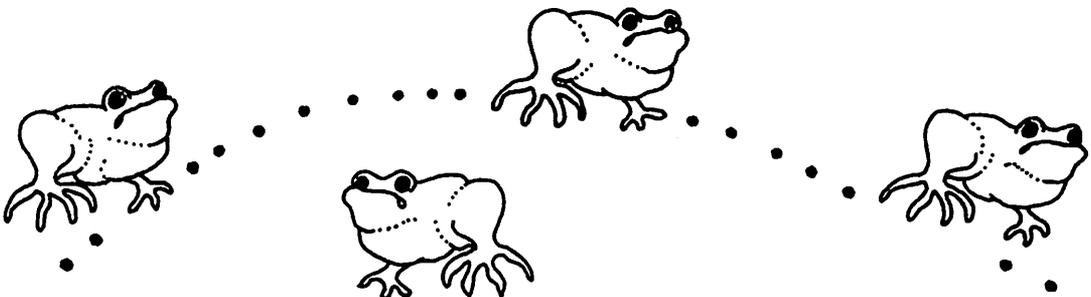
```
! " # $ % & '
( ) * + , - . /
0 1 2 3 4 5 6 7
8 9 : ; < = > ?
@ ABCDEFG
H I J K L M N O
P Q R S T U V W
X Y Z [ \ ] ^ _

**DONE**
>□
```

Hold on just a minute! There are only sixty-three characters on the screen! What happened to the other one? Well, there *are* actually sixty-four. Look at the top line, and notice that it appears to be indented one space. But remember, the first CHAR value we assigned was 32. *32 is a space*. Even though a space doesn't print anything on the screen, it does occupy room on a line, and it *is* a character, as far as the computer is concerned.

Animation

Animation is the illusion of movement. In order to achieve this illusion in your graphics programs, it's necessary to change your character or sets of characters periodically. The following programs demonstrate some of the techniques used to create flashing and moving graphics on the screen.



Flashing Letters

One way to create a flashing graphic is to print a character (or set of characters), delay the program, clear the screen, delay the program again, and then repeat the process. The clearing of the screen and the delays have the effect of turning the character "on and off," making it appear to flash.

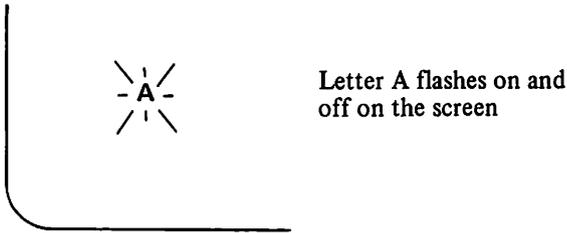
Let's try a program that flashes the letter A in the center of the screen.

```

NEW
10 CALL VCHAR(12,16,65) ← Character "on"
20 GOSUB 100
30 CALL CLEAR ← Character "off"
40 GOSUB 200
50 GO TO 10
100 FOR DELAY=1 TO 300
110 NEXT DELAY
120 RETURN
200 FOR DELAY=1 TO 200
210 NEXT DELAY
220 RETURN
  
```

} Separate time delays –
different delay count

Now clear the screen and run the program. (Press **SHIFT C** to stop the program.)



Another way to simulate flashing is to replace one character with another in the same spot on the screen. Let's revise our program so that it alternately flashes A and I. We can do this easily by entering a new line 30:

```

same position
  ↙ ↘
30 CALL VCHAR(12,16,73)
  ↑
code for I
  
```

Since we're replacing A with I, we don't have to clear the screen between printing the characters. However, we may want to add a **CALL CLEAR** at the beginning of the program. So enter this line:

```

5 CALL CLEAR
10 CALL VCHAR(12,16,65)
20 GOSUB 100
30 CALL VCHAR(12,16,73)
40 GOSUB 200
50 GOTO 10
  
```

Run the revised program. Do A and I appear to flash alternately on the screen? (You may want to increase the time delay in subroutine 200 so that A and I will each stay on the screen the same length of time. A better way to obtain equal time delays would be to use the same delay subroutine and go to that subroutine twice. Just change line 40 to GOSUB 100.

40 GOSUB 100

Then you don't need subroutine 200 at all. From flashing characters to flashing color squares is an easy step. Next we'll examine a program that places a flashing color square on the screen.

Flashing Color Squares

With this program we want to create a color square that flashes on the screen. We'll write the program so that we can input the color we want, and we'll use character 42 (the asterisk, in character set 2) to make our square.

```

NEW
10 CALL CLEAR
20 INPUT "COLOR CODE?":X
30 CALL CLEAR
40 CALL COLOR(2,X,X)
50 CALL VCHAR(12,16,42)
60 N=500
70 GOSUB 200
80 CALL CLEAR
90 N=300
100 GOSUB 200
110 GO TO 40
200 FOR B=1 TO N
210 NEXT B
220 RETURN

```

← Accept color code (1–16)
 ← Clear screen
 ← Define color for character set 2, which contains the asterisk, and make foreground and background the same color
 } ← New way – same time delay
 } ← N = TIME
 ← N has been assigned in main program

Now run the program. First, it asks

COLOR CODE?

and waits for you to input a valid color code. The codes are 1 through 16; remember, however, that code 1 is transparent and code 4 is the normal screen color in the RUN Mode. Squares of these colors will not show up on the screen.

When you type in a color code and press **ENTER**, you'll see the square flashing near the center of the screen. (Press **SHIFT C** to stop the program.) Try the program with several several color codes.

Next, let's change the program to create *two* color squares that alternately flash on the screen. To do so, we'll need to input two color codes. So enter these lines first:

```

20 INPUT "COLOR 1?":X
25 INPUT "COLOR 2?":Y

```

← Accept color for first square
 ← Accept color for second square

Now we'll replace our original line 80 with two new lines, to set the color and display the second square:

```

80 CALL COLOR(2,Y,Y)      Set color for second square
85 CALL VCHAR(12,16,42)

```

Let's review these changes by listing the program. Clear the screen; then type LIST and press ENTER:

```

LIST
10 CALL CLEAR
20 INPUT "COLOR 1?":X
25 INPUT "COLOR 2?":Y
30 CALL CLEAR
40 CALL COLOR(2,X,X)      }
50 CALL VCHAR(12,16,42)  } First color square
60 N=300                  } is on the screen
70 GOSUB 200
80 CALL COLOR(2,Y,Y)      }
85 CALL VCHAR(12,16,42)  } Second color square
90 N=200                  } is on the screen
100 GOSUB 200
110 GOTO 40               ← Repeat
200 FOR B=1 TO N
210 NEXT B
220 RETURN

```

Select your two colors and run the program, typing in the color codes as the program asks for them. The two color squares will alternately flash on the screen.

Experiment with several color combinations to find those that give a good contrast. Here are a few examples to try:

COLOR 1	COLOR 2
6	5
11	14
14	16
9	11

Moving Color Squares

With just a few simple changes in the previous program, we can make the color squares move across the screen as they flash. Add these lines:

```

26 FOR K=3 TO 28
50 CALL VCHAR(12,K,42)
85 CALL VCHAR(12,K,42)
105 NEXT K
110 GO TO 10

```

Starting at column 3, the squares flash and travel across the screen, ending at column 28. Then the screen clears, and the program asks you for the new color inputs. Run the program and see the effects.

Here is the listing of our final revision of the flashing squares program.

```

LIST
10 CALL CLEAR
20 INPUT "COLOR 1?":X
25 INPUT "COLOR 2?":Y
26 FOR K=3 TO 28
30 CALL CLEAR
40 CALL COLOR(2,X,X)
50 CALL VCHAR(12,K,42)
60 N=300
70 GOSUB 200
80 CALL COLOR(2,Y,Y)
85 CALL VCHAR(12,K,42)
90 N=200
100 GOSUB 200
105 NEXT K
110 GO TO 10
200 FOR B=1 TO N
210 NEXT B
220 RETURN

```

If you want to speed up the flashing, shorten the time delay setting in lines 60 and 90. For a challenge, you might like to make the program flash *three* color squares! How would you do it?

By this time you've seen several examples of the kind of graphics you can create with the standard characters of your computer. Later, you'll see how to develop your own characters.

Error Conditions with FOR-NEXT

We mentioned earlier that a nested loop *must* be completely contained within another loop. If your program included lines like these:

```

10 FOR A=1 TO 6
30 FOR X=5 TO 10

80 NEXT A
90 NEXT X ← Should be "nested" within the "A" loop

```

the computer would stop the program and give you this error message:

```
* CAN'T DO THAT IN 90
```

The computer can't go back inside the completed "A" loop to pick up the beginning of the "X" loop.

Another possible error condition with FOR-NEXT statements is accidentally omitting either the FOR line or the NEXT line. For example, if you attempted to run this program:

```
10 FOR A=1 TO 5
20 PRINT A
30 END
```

the computer would respond with

*** FOR-NEXT ERROR**

If you encounter an error message, just list the program (type LIST and press ENTER), identify the error, and correct the problem line or lines.

Chapter Summary

FOR-NEXT	You've used this statement to build controlled loops that repeat a part of the program a specified number of times or create a time delay in the program.
GOSUB	You've placed parts of programs that will be repeated in a sub-routine to be "called" when needed by the GOSUB statement.
RETURN	You've learned to use FOR-NEXT with COLOR, VCHAR and HCHAR to create designs and even move them around on the screen.

Try the exercises that follow to refresh your memory and review all that you have learned in this chapter.

Chapter Five Exercises

- (1) In the following FOR-NEXT loop, give the value of A that would be printed by line 30. _____

```

10 FOR A=1 TO 21
20 NEXT A
30 PRINT A
    
```

- (2) This program displays a color bar. Tell why the time delay is desirable.

```

10 CALL CLEAR
20 FOR A=1 TO 16
30 CALL COLOR(1,A,A)
40 CALL HCHAR(9,3,36,24)
50 FOR B=1 TO 400
60 NEXT B
70 NEXT A
    
```

The time delay _____

- (3) Complete this sentence.

In exercise 2, the FOR-NEXT loop (lines 50–60) is said to be _____ inside the other FOR-NEXT loop (lines 20–70);

- (4) Re-write the program in Exercise 2 so that the time delay is placed in a subroutine starting at line 100.

```

10 _____
20 _____
30 _____
40 _____
50 _____
60 _____
70 _____
110 _____
120 _____
130 _____
    
```

- (5) Can the same subroutine be called from more than one place in the main program?

- (6) This program is a variation of COLOR BAR TWO.

```

10 FOR A=1 TO 16
20 GOSUB 200
30 CALL HCHAR(A+3,3,42,28)
40 GOSUB 300
50 NEXT A

60 FOR A=1 TO 16
70 GOSUB 200
80 CALL VCHAR(1,A+5,42,24)
90 GOSUB 300
100 NEXT A
110 END

```

Write the subroutines which will make the output display appear the same as in the original program.

```

200 _____
210 _____
220 _____
300 _____
310 _____
320 _____

```

- (7) The character codes for the letters of the alphabet run from 65 through 90. Complete this program which flashes each alphabetic character (in order) at a position in the 12th row and 24th column.

```

10 CALL CLEAR
20 FOR X= _____
30 CALL VCHAR(_____,_____)
40 FOR B=1 TO 300
50 NEXT B
60 _____

```

- (8) Complete this sentence.

To make a solid square of color with the CALL COLOR statement, you make the _____ color code the same as the _____ color code.

- (9) Complete this sentence.

The color code values range from _____ to _____.

- (10) Complete this program which will display color squares in a diagonal line from position : row 3, column 3 to position: row 15, column 15.

```

10 CALL CLEAR
20 FOR X= _____
30 CALL COLOR(2,7,7)
40 CALL VCHAR(____,____,42)
50 NEXT _____
60 FOR B=1 TO 500
70 NEXT B
80 END

```

Answers to Chapter Five Exercises

- (1) 22 would be the value for A after exiting the loop.
- (2) The time delay keeps the color bar on the screen long enough to see it.
- (3) nested
- (4)

```

10 CALL CLEAR
20 FOR A=1 TO 16
30 CALL COLOR(1,A,A)
40 CALL HCHAR(9,3,36,24)
50 GOSUB 100
60 NEXT A
70 END
100 FOR B=1 TO 400
110 NEXT B
120 RETURN

```
- (5) yes
- (6)

```

200 CALL CLEAR
210 CALL COLOR(2,A,A)
220 RETURN
300 FOR B=1 TO 200
310 NEXT B
320 RETURN

```
- (7)

```

10 CALL CLEAR
20 FOR X=65 TO 90
30 CALL VCHAR(12,24,X)
40 FOR B=1 TO 300
50 NEXT B
60 NEXT X

```
- (8) make the *foreground* color code the same as the *background* color code.
- (9) from 1 to 16
- (10)

```

10 CALL CLEAR
20 FOR X=3 TO 15
30 CALL COLOR(2,7,7)
40 CALL VCHAR(X,X,42)
50 NEXT X
60 FOR B=1 TO 500
70 NEXT B
80 END

```

Chapter Six

Beginning Simulation

As you progress through the book, your programs will grow longer and longer. After the program has been laid aside for awhile, it becomes more difficult to follow the flow of executions. You may find you need some way to organize and document the sections of longer programs.

The REMARK statement allows you place notes in the program that tell what each section of a program does. REMARK statements are not executed but are inserted to make programs clear and understandable.

For example, here is a program repeated from Chapter Five with REMARK (abbreviated, REM) statements inserted.

Abbreviation for REMARK

```
10 REM ** INPUT COLORS TO BE USED **
20 CALL CLEAR
30 INPUT "COLOR 1?": X
40 CALL CLEAR
50 INPUT "COLOR 2?": Y
60 CALL CLEAR

70 REM ** PUT FIRST COLOR ON **
80 CALL COLOR(2,X,X)
90 CALL VCHAR(12,16,42)
100 N=300
110 GOSUB 200

120 REM ** PUT SECOND COLOR ON **
130 CALL COLOR(2,Y,Y)
140 CALL VCHAR(12,16,42)
150 N=200
160 GOSUB 200

170 REM ** REPEAT IT ALL **
180 GO TO 70

200 REM ** TIME DELAY SUBROUTINE **
210 FOR B = 1 TO N
220 NEXT B
230 RETURN
```

The REM lines are not executed

Even though your program gets longer, you will find that REMARK statements are worthwhile. We will use them frequently from now on.

The INTeger Function

The INT function gets its name from the word *integer*. Integers include zero and all those positive and negative numbers that have no digits after the decimal point.

The best way to learn how the INT function works is by trying it. First, let's work a division problem that doesn't result in a whole number answer.

Type:

```
PRINT 16/3
```

and press ENTER. The answer is 5.333333333

Now try this example:

```

>PRINT INT(16/3)
5
>□

```

INT kept the whole number part of the answer and threw away the digits after the decimal point.

Try another example:

```

PRINT INT(7/6) ← 7/6 = 1.666666666
1                INT(7/6) = 1

```

The answer is 1; all of the fractional part has been discarded.

How about a real-life problem? Let's say a sales clerk is giving \$1.37 in change to a customer. The customer wants as many quarters as possible. How many quarters can be given?

```

PRINT INT(1.37/.25)
5

```

The answer is 5. Five quarters can be given.

You can also put more than one instance of INT in a PRINT statement.

Type: `PRINT INT(7/3);INT(10/3)`
Press ENTER

Again the whole number
parts of $7/3$ and $10/3$

```
>PRINT INT(7/3);INT(10/3)
  2 3
>□
```

What happens if the number is less than one?

Type: `PRINT INT(1/3)`
Press ENTER

$1/3 = 0.333333\dots$
whole number part
is zero

```
>PRINT INT(1/3)
  0
>□
```

What would happen if you put the following values in the INT function: 8.99, 8.56, 8.01?

Try them and see.

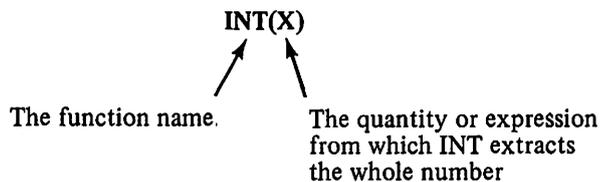
Type: `PRINT INT(8.99);INT(8.56);INT(8.01)`
Press ENTER

All the same

```
>PRINT INT(8.99);INT(8.56);INT(8.01)
  8 8 8
>□
```

INT produces the same result in the last examples no matter what digits are after the decimal point. It *does not* round the number off to the nearest integer.

The general form of the INT function is:



To see better how INT works, try the following program.

```
10 REM ** INT EXPERIMENT *
20 CALL CLEAR
30 FOR A = 1 TO 10
40 PRINT A/3, INT(A/3)
50 NEXT A
```

Type: NEW
Enter the program
RUN

```
.333333333 0
.666666666 0
1          1
1.333333333 1
1.666666666 1
2          2
2.333333333 2
2.666666666 2
3          3
3.333333333 3
```

} Only integer part of result

```
**DONE**
>□
```

From the results of this program, you *could* say that the INTeger function merely throws away all that part of a number following the decimal point. But don't be too hasty. So far, you have looked only at positive numbers. What happens with *negative* numbers?

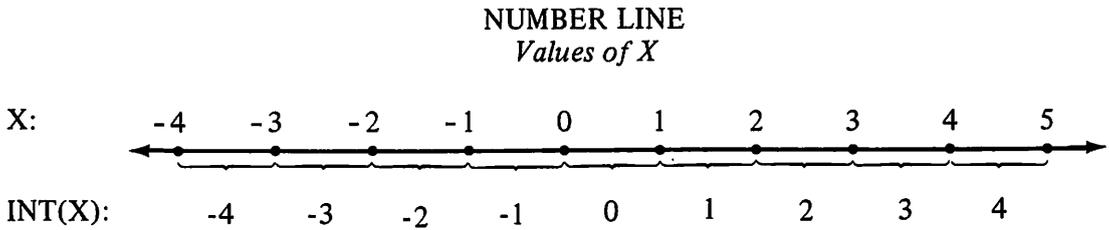
We'll use a program to explore INT and negative numbers. **ENTER** these lines:

```
NEW
10 REM**NEGATIVE INTEGERS**
20 CALL CLEAR
30 FOR A=1 TO 7
40 PRINT -A/3,INT(-A/3)
50 NEXT A
60 END
```

Now run the program. The screen will show these results.

```
-.333333333 -1
-.666666666 -1
-1          -1
-1.333333333 -2
-1.666666666 -2
-2          -2
-2.333333333 -3
```

So $\text{INT}(X)$ - where X represents a number or a mathematical expression - computes the nearest integer *that is less than or equal to* X . Perhaps looking at a *number line* will help to explain what happens.



As can be seen from the number line, when X has the value $-1/3$, the largest integer that is still smaller than X is -1 . The INT function is often referred to as “the Greatest Integer” function for this reason. Thus $\text{INT}(-1/3)$ gives the result -1 , as shown on the screen.

One last feature associated with INT is very useful to know. It can appear on the right side of an equal sign in a LET statement. For example, try the next series of lines.

```
> LET A=INT(4/3)+2
> PRINT A
3
>□
```

In the LET statement, $\text{INT}(4/3)$ produces the integer result of 1. This result is added to the constant 2, yielding 3 as a final result. A is then assigned the value of 3 and printed.

Several applications of the INT function are shown later in this chapter. For now, try some other experiments with INT so that you become even more familiar with how it works. Can you round off money to the nearest penny? What is $4/3$ of a dollar expressed in dollars and cents?

```
> LET A=INT(100*4/3+.5)
> PRINT A/100
1.33
>□
```

↖ What's this for?

How did that work?

1st $100 * 4/3+.5= 133.3333333+.5= 133.8333333$
 2nd $INT(100 * 4/3+.5) = 133$
 3rd $133/100= 1.33$ ← One dollar and thirty three cents (to the nearest penny)

Have you discovered why the .5 is added to the computation? Try looking at 5/3 dollars in terms of dollars and cents. What happens if you don't add .5? The result is *not* to the *nearest cent*!

$100 * 5/3 = 166.6666666$
 $INT(100 * 5/3) = 166$
 $166/100 = 1.66$ ← Not 1.67!

But $100 * 5/3+.5 = 166.6666666+.5 = 167.1666666$
 $INT(100 * 5/3+.5) = 167$
 $167/100 = 1.67$ ← Correct to the *nearest cent*

The RND Function

Many computer programs are “simulations” that imitate some real event. For example, a computer program that imitates the rolling of a pair of dice is a simple simulation. We'll develop, enter, and run a dice-rolling program. Other programs that are included will explore the games, graphics, and musical capabilities of your computer.

The heart of most of these games and simulations is the RND function, so let's begin there.



This is a *die*



These are two *dice*

The letters in the name RND are taken from the word RaNDom. To discover exactly how RND works in your Home Computer, first try this function in the IMMEDIATE Mode. To start, erase whatever old program may be in your Home Computer and CLEAR the screen. Do you recall how this is done?

Type: NEW
 CALL CLEAR

>□

Now enter the following command.

Type: **PRINT RND**

```

>PRINT RND
.5291877823 ← RND produced this number
>□
  
```

Try the same statement again!

Type: **PRINT RND**

```

>PRINT RND
.5291877823
>PRINT RND
.3913360723 ← The numbers are not the same
>□
  
```

Here is an interesting situation! Every time we use RND, we get a different number. The two numbers we show above are different, and the results you obtained when you tried it were different too. That's exactly what RND does – it generates “random numbers,” numbers that do not follow any order or sequence.

Type: **PRINT RND,RND**

```

>PRINT RND,RND
.5343438556 .3894551053
>□ ← Two on a line
  
```

Here is a program that will produce ten random numbers using RND.

```

20 FOR COUNT = 1 TO 10
30 PRINT RND
40 NEXT COUNT
  
```

If you enter the program into your Home Computer and RUN the program, a list of ten random numbers is produced on your screen.

Type: RUN

```

>RUN
.5291877823
.3913360723
      :
      :
.7000201301
.0010849577
**DONE**

>□
  
```

} 10 random numbers

You may have noticed that all the numbers that RND generates are less than one (1.0) in value. Also, there are no negative numbers. RND is preset to produce only numbers that are greater than or equal to zero (0). Everytime your Home Computer sees RND in a program it generates a random number between zero and one. In the program given above, RND was used 10 times and produced 10 random numbers.

Make a note of the first few random numbers in the list on your screen. Now RUN the program again.

Type: RUN

```

>RUN
.5291877823
.3913360723
      :
      :
.7000201301
.0010849577
**DONE**

>□
  
```

} They are the same as last time

The list of numbers is the same! This feature of the RND function is important to remember and can be used to advantage in some applications. Within a program RND will produce the same sequence of random numbers each time the program is RUN.

UNLESS . . . !

Unless the BASIC language command **RANDOMIZE** is used in your program. Add the **RANDOMIZE** statement, shown below, to the program that is still in your Home Computer.

10 RANDOMIZE

Your altered program now looks like this:

```
5 REM*10 RANDOM NUMBERS*
10 RANDOMIZE
20 FOR COUNT = 1 TO 10
30 PRINT RND
40 NEXT COUNT
```

Now **RUN** this altered program.

Type: **RUN**

```
>RUN
.1176354017 ← New list of 10 random numbers
.5120893264
:
:
.3207659957
.975423765
**DONE**

>□
```

Your numbers may not be the same as these

The ten numbers in this list are not the same as those in the previous list. **RAN** **RANDOMIZE** has caused the function **RND** to produce a new list. To check that this occurs each time, mark down the first few numbers in the current list and **RUN** the program again.

Type: **RUN**

```
>RUN
.0765398493 ← A totally new list of 10 numbers
.5523019627
:
:
.7127096875
.3884785113
**DONE**

>□
```

How do the two lists compare? They should be quite different due to the RANDOMIZE statement.

You might want to experiment with this program for a while. Delete line 10 and RUN the program a few times. Put line 10 back in and RUN it several times. Understanding RND and RANDOMIZE makes it easy to begin to build programs for games and simulations.

This 

```
5 REM*10 RANDOM NUMBERS*
20 FOR COUNT = 1 TO 10
30 PRINT RND
40 NEXT COUNT
```

Then this 

```
5 REM*10 RANDOM NUMBERS*
10 RANDOMIZE
20 FOR COUNT = 1 TO 10
30 PRINT RND
40 NEXT COUNT
```

The preceding discussion explained what the RND function does – produces “random numbers” from zero but always *less than* one.

Now let us examine how random numbers outside the range of zero to one can be generated.

The RND function can be used as part of any legitimate computation. For example, $10*\text{RND}$ or $10*\text{RND}+7$ are both legitimate uses of RND within BASIC. To show what is produced when RND is used in this way, try the following on your Home Computer.

Type: **CALL CLEAR**
PRINT 10*RND

```
>PRINT 10*RND
5.765389224
>□
```

 You may show a different number

Try this statement several times. Is it clear to you that RND, when used in this manner, produces “random numbers” from zero (0) to 9.9999... inclusive?

0, .0000000001, .0000000002, . . . , 9.999999999, 10
0 to 9.999999999 inclusive

As an aid in seeing that this last comment is true, change line 30 of the program in your Home Computer. Replace the current line 30 with the following entry:

```
30 PRINT 10*RND
```

Your program now looks as follows:

```
5 REM * 10 RANDOM NUMBERS *
10 RANDOMIZE
20 FOR COUNT = 1 TO 10
30 PRINT 10*RND
40 NEXT COUNT
```

Now RUN your altered program.

Type: RUN

```
>RUN
 3.796523017
 1.117089595
      .
      .
.6517942135
2.817156306
** DONE **
>□
```

} 10 random numbers,
0 ≤ N < 10

So, to produce a set of “random numbers” from zero to 9.99999 . . . (inclusive) multiply RND by ten. If you want numbers that are greater than zero but less than 100, then you would multiply RND by one hundred (100*RND).

It becomes difficult to avoid technical “jargon” at this point. The word, between, used literally means all those numbers between the end points but not including them. “Between 0 and 10” excludes 0 and 10 (As in Joe is between Suzie and Mike). But “from 0 to 10, inclusive” includes the end points 0 and 10. The mathematical symbolism used above ($0 \leq N < 10$) is much more precise. It includes the end point 0, but *not* the end point 10.

As can be seen on the screen, the numbers produced in this fashion have digits after the the decimal point. For many applications it is useful to be able to access and use only the integer portion of the number (the digits to the left of the decimal point). This can be accomplished by combining two BASIC functions, INT and RND. The INT function was introduced in the preceding section. Since RND can be part of any valid computation, it can be used inside the INT function. Try the following statement on your Home Computer.

Type: **CALL CLEAR**
PRINT INT(10*RND)

```

>PRINT INT(10*RND)
 6
>□

```

You may show a different number. After all, it *is* random!

Type in the same statement several times and observe the sequence of numbers produced.

All numbers will have integer values from zero to nine. INT throws away the digits to the right of the decimal point and keeps those to the left. The following table describes how INT, RND, and $10*RND$ work together.

	<u>RND</u>	<u>10*RND</u>	<u>INT(10*RND)</u>
PRODUCES NUMBERS IN RANGE	0 to .99999 ...	0 to 9.99999 ...	0 to 9 Integers only

Change line 30 in the program in your Home Computer to read as follows:

```
30 PRINT INT(10*RND)
```

Your altered program now appears this way.

```

5 REM * 10 RANDOM INTEGERS *
10 RANDOMIZE
20 FOR COUNT = 1 TO 10
30 PRINT INT(10*RND)
40 NEXT COUNT

```

RUN the program.

```

>RUN
 5
 7
 8
 .
 .
 0
 8

```

Ten random integers.
All of them are in the range, 0 through 9

```

** DONE **
>□

```

In some games and simulations, you will need random numbers that begin at some value other than zero. For example, to simulate the throw of one die you need a random generator that produces values from one to six. You have seen that `INT(10*RND)` gives values from zero to nine. What would `INT(6*RND)` produce? Change line 30 in the program to `PRINT INT(6*RND)` and `RUN` the new program.

Type: `30 PRINT INT(6*RND)`
`RUN`

```

>RUN
 4
 0
 1
 .
 .
 .
 1
 3
} All values between 0 and 5

** DONE **

>□

```

Notice that some integers may appear more than once.

Your screen shows a list of ten “random numbers” from 0 to 5. What would happen if we added the value 1 to each item in this list? The resultant numbers would range from 1 to 6. That’s just what we need to simulate the throw of a single die. Again, alter the program as shown below and run it.

Type: `30 PRINT INT(6*RND)+1`
`CALL CLEAR`
`RUN`

```

>RUN
 3
 4
 2
 .
 .
 .
 6
 2
} Simulation of 10 rolls of a die

** DONE **

>□

```

Congratulations! The program now in your Home Computer is a simulation (imitation) of throwing a single die ten times.

Here's what it looks like.

```
5 REM * 10 RANDOM INTEGERS *
10 RANDOMIZE
20 FOR COUNT = 1 TO 10
30 PRINT INT(6*RND)+1
40 NEXT COUNT
```

A Two-Dice Simulation

At this point you can easily design a program to simulate the throws of two six-sided dice. Before you start, erase the old program by typing NEW.

Enter the following program:

```
10 REM * ROLL 2 DICE *
20 CALL CLEAR
30 RANDOMIZE
40 INPUT "NUMBER OF ROLLS: " :ROLLS ← Accept number of throws to simulate
50 REM * ROLL 'EM *
60 FOR COUNT = 1 TO ROLLS
70 DIE1 = INT(6*RND)+1 } ← Simulate a throw
80 DIE2 = INT(6*RND)+1 }
90 PRINT DIE1;DIE2;DIE1+DIE2 ← Display each throw and sum of "spots"
100 NEXT COUNT
110 PRINT
120 REM * ANOTHER ROUND *
130 GO TO 40
```

This program simulates the rolling of two dice and prints out the result of each roll and the sum of the spots on the dice faces. You are asked how many rolls you wish to make at the start of the program. RUN the program and watch what occurs.

Type: RUN

```
NUMBER OF ROLLS:  ← Cursor is here
```

The program prints a request for the number of rolls to make. Enter a number and press the **ENTER** key.

Type: 5 (for example)

```

>RUN
NUMBER OF ROLLS: 5
  2  5  7
  6  6 12
  3  1  4 ← Your screen shows 5 throws
  2  3  5
  1  4  5
NUMBER OF ROLLS: □

```

The program keeps looping back to the INPUT request line. Typing a **SHIFT C** stops the program. Enter different values for the number of rolls. What happens if you try 30 rolls? Make some changes to the program. For example, how would you alter the program to simulate the throwing of two eight-sided dice?

RND can be used in other interesting ways. For example, with RND you can explore both the graphical and musical capabilities of your Home Computer.

Earlier, you saw the results of using the graphics routines VCHAR and HCHAR. With these routines, a single character could be placed anywhere on the screen. Let us now look at a program that randomly places characters on the screen.

Erase your old program and type this one.

```

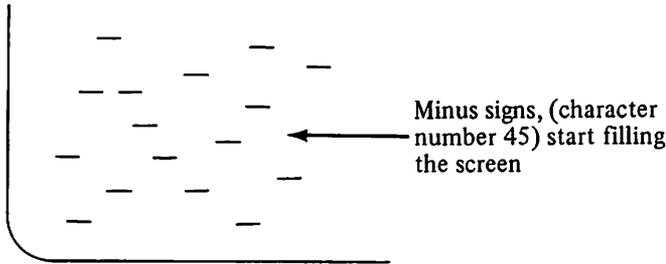
Type: 5 REM**RANDOM CHARACTERS**
      10 RANDOMIZE
      20 INPUT "CHARACTER NUMBER" :CHARNUM ← Accept character number
      30 CALL CLEAR
      40 REM**PRINT IT RANDOMLY**
      50 COLUMN = INT(32*RND)+1 ← Pick a random column
      60 ROW = INT(24*RND)+1 ← Pick a random row
      70 CALL VCHAR(ROW,COLUMN,CHARNUM) ← Display character
      80 GO TO 50

```

The VCHAR routine needs two values to tell it where to place the character on the screen. The two values correspond to a position in a row and column of the screen face. Across the screen, there are 1 to 32 column positions: down the screen, there are 1 to 24 rows. Lines 50 and 60 of the program provide random values along each of these dimensions of the screen. The input to the program is a character number. The valid character numbers are 32 to 95 (see Appendix A).

RUN the program and try an arbitrary valid character number.

Type: RUN
CHARACTER NUMBER? 45



To stop the program, type in a **SHIFT C**. Try several different characters to see if any produce unusual designs.

The concepts in the program just presented could be used as part of a larger program that requires the random placement of characters on the screen before the start of a game. Notice that each time you run the program, the initial character almost never appears in the same screen positions. If the **RANDOMIZE** statement is taken from the program, the initial character as well as the overall design is always the same. You might remove line 10 from the program and verify this result for yourself.

When you've finished experimenting with different characters, let's change the program to generate characters at random, as well as place them randomly on the screen. Now, first we'll have to decide how to set the limits we want for the character range. We know that there are 64 characters, with character codes ranging from 32 through 95, and we know that we do *not* want to use character code 32 (a blank space). How about:

$$\text{INT}(63 * \text{RND}) + 33$$

\uparrow \uparrow
 64 - 1 32 + 1

to generate our character number? $\text{INT}(63 * \text{RND})$ will produce random integers from 0 through 62. Now check the limits established when we add 33:

$$0 + 33 = 33 \quad \text{— lowest possible character code}$$

$$62 + 33 = 95 \quad \text{— highest possible character code}$$

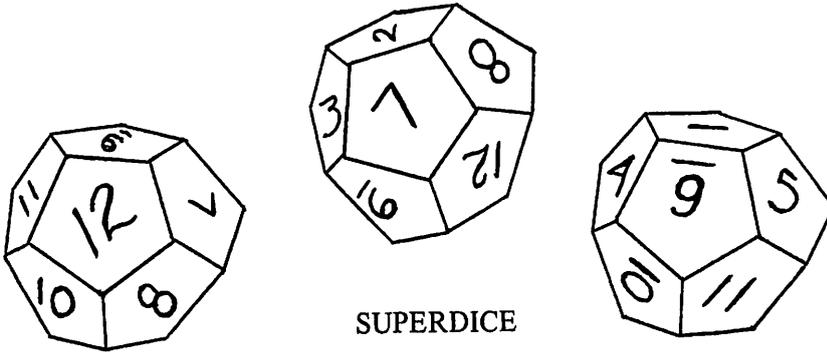
Clear the machine and enter this new program:

```

5 CALL CLEAR
10 RANDOMIZE
20 REM * RANDOM CODE *
30 CODE=INT(63*RND)+33
40 REM * RANDOM PLACEMENT *
50 ROW=INT(24*RND)+1
60 COLUMN=INT(32*RND)+1
70 CALL VCHAR(ROW,COLUMN,CODE)
80 REM * BACK FOR MORE *
90 GO TO 30
  
```

Now run the new program. This time, you'll see different characters appearing in random positions on the screen. (Press **SHIFT C** when you're ready to stop the program.)

Random character placement can be used very effectively in graphics programs. We'll explore a couple of examples later in this book, but for now let's look at some possible RND error conditions.



Error Conditions with RND

The error messages produced by an improper usage of RND are essentially the same as the error messages we've mentioned before. Here are some examples:

Typing Errors

PRINT RND
 10 PRINT RND
 10 PRINT INT(10*RND

Should be RND

Missing close parenthesis

Error Message

{ No error message is given. The computer assumes that RND is a variable name and prints the value assigned to that variable. If no value has been assigned, it prints zero (0) on the screen.
 *INCORRECT STATEMENT IN 10

About the only new error condition we need to mention occurs if you try to use the letters RND as a numeric variable name in a LET or assignment statement. For example, if you type:

```
LET RND=5
```

the computer will respond with

```
* INCORRECT STATEMENT
```

This occurs because RND can be used only as a function in TI BASIC. It must be on the right side of the equal sign as in this example.

```
LET A=RND
```

Chapter Summary

In this chapter you have discovered some statements that help you create random events for graphics and simulations. You have also found a tool that aids you in making your programs clear and organized.

REM	The REMark statement allows you to annotate your programs with explanatory notes and comments.
INT	You have learned to use the INTeger function to convert decimal numbers to integers.
RND	You have learned to create numbers in a random fashion with the RND statement.
RANDOMIZE	You have learned how to always get a new series of random numbers by using the RANDOMIZE statement at the beginning of a program.

All of these statements have been used to increase the power of your programming ability. Check what you have learned by doing the following exercises.

Chapter Six Exercises

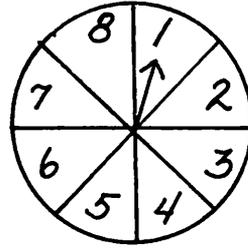
- (1) Are REMark statements executed by the computer? _____
- (2) Why are REMark statements used in a program? _____

- (3) What number would be printed following this statement in the Immediate Mode?
PRINT INT(19/3) _____
- (4) Which of the following statements would produce the largest result? _____
- (a) PRINT INT(25/5)
(b) PRINT INT(27/5)
(c) PRINT INT(26/5)
(d) PRINT INT(28/5)
- (5) What is the value of INT(-6.32)? _____
- (6) What would be printed when the following program is executed?

```
100 CALL CLEAR
110 A = 2*3.3333333
120 B = INT(100*A)/100
130 PRINT B
```

- (7) Write a short program to print 10 random numbers in the range of 3 to 9. (Not including 9)
- _____
- _____
- _____
- _____
- (8) Rewrite your program of exercise 7 to make the numbers *integers* from 0 through 9 inclusive (0,1,2,3,4,5,6,7,8, or 9).

- (9) A wheel with a spinning pointer has 8 numbers on its face as pictured below. Write a program to simulate the results of spinning the pointer (10 spins).



- (10) Write a program that will place random letters of the alphabet, A through G, in random locations on the screen.

Answers to Chapter Six Exercises

- (1) No
- (2) To make a program clear and understandable.
- (3) 6
- (4) a, b, c, and d all give 5 for a result.
- (5) -7
- (6) 6.66
- (7) One way:

```
10 CALL CLEAR
20 FOR N = 1 TO 10
30 PRINT 6*RND + 3
40 NEXT N
```
- (8) One way:

```
10 CALL CLEAR
20 FOR N = 1 TO 10
30 PRINT INT(10*RND)
40 NEXT N
```
- (9) One way:

```
10 CALL CLEAR
20 FOR N = 1 TO 10
30 PRINT INT(8*RND)+1
40 NEXT N
```
- (10) One way:

```
10 CALL CLEAR
20 RANDOMIZE
30 A = INT(7*RND)+65 ← Gives character code 65-71 (A through G)
40 ROW = INT(24*RND)+1
50 COL = INT(32*RND)+1
60 CALL VCHAR(ROW,COL,A)
70 GO TO 30
```

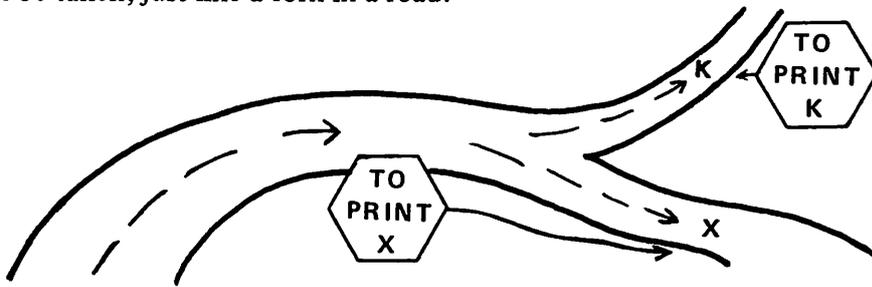
Chapter Seven

More Program Control Statements

Up to this point most of the programs have been constructed so that they either run straight through or loop using a FOR-NEXT loop or perhaps a GO TO statement. We have also used subroutines for time delays. In this chapter we'll introduce several versatile statements that allow changing the order of normal sequential execution.

The IF-THEN Statement

The IF-THEN statement provides you with the capability of making branches or “forks” in your program. A branch or fork is a point in a program where either one of two paths can be taken, just like a fork in a road.



The general form of an IF-THEN statement looks like this:

IF *condition* **THEN** *line number*

The “condition” is the true/false relationship between two BASIC expressions. The “line number” is the program line that you want to “branch” to if the “condition” is true. If the “condition” is *not true*, then the program line following the IF-THEN statement is executed. For example:

```
80 IF K < 10 THEN 50
```

The statement says: If the value of K is less than 10, then go to line 50 of the program. If K is greater than (>) or equal to (=) 10, then do not branch to line 50. Instead, execute the line following line 80.

Let's try a demonstration program. Enter these lines:

```

NEW
10 REM**IF-THEN DEMONSTRATION PROGRAM**
20 CALL CLEAR
30 REM**K IS A COUNTER**
40 LET K=1
50 PRINT "K=";K
60 LET K=K+1
70 REM**TEST FOR K LESS THAN 10**
80 IF K<10 THEN 50
90 REM**PROGRAM GOES HERE WHEN K=10**
100 PRINT "OUT OF LOOP"
110 END

```

If new value of K is less than 10,
go back to line 50 and repeat

If new value of K
is *not* less than 10,
go on to next line

Now run the program.

```

K= 1
K= 2
K= 3
K= 4
K= 5
K= 6
K= 7
K= 8
K= 9
OUT OF LOOP
**DONE**


```

Each time the program reaches line 80, it must make a true or false decision. When K is nine or less, the IF condition ($K < 10$) is true and the program branches to line 50. When K equals 10, however, $K < 10$ is false. The program then executes line 100 and stops.

Directionally, you can consider the statement in this way:

If true, follow this path 

```

80 IF K<10 THEN 50
    ↑           ↑
Condition   Line number

```

If false, follow this path
to the next statement



A trace of the program.

Statement	K	Remarks
20 CALL CLEAR	0	CLEAR the screen
40 LET K=1	1	Assign initial value
50 PRINT "K=";K	1	Display K
60 LET K=K+1	2	Add 1 to K
80 IF K<10 THEN 50	2	K=2; less than 10
50 PRINT "K=";K	2	Display K
60 LET K=K+1	3	Add 1 to K
80 IF K<10 THEN 50	3	K=3; less than 10
50 PRINT "K=";K	3	Display K
:	:	Repeats until K=9
:	:	
50 PRINT "K=";K	9	Display K
60 LET K=K+1	10	Add 1 to K
80 IF K<10 THEN 50	10	K=10; is <i>not</i> less than 10
100 PRINT "OUT OF LOOP"	10	PRINT message
110 END	10	STOP program

We mentioned earlier that the condition is true/false relationship between two expressions. In the example just seen, the relationship was $<$, or "less than." There are six relationships that can be used in the IF-THEN statement:

Relationship	Math Symbol	BASIC Symbol
Equal to	=	=
Less than	<	<
Greater than	>	>
Less than or equal to	\leq	<=
Greater than or equal to	\geq	>=
Not equal to	\neq	<>

Suppose we changed line 80 of the program to this:

80 IF K<= 10 THEN 50

How would the program's performance be affected? Try it! Enter the new line, then run the program again.

```

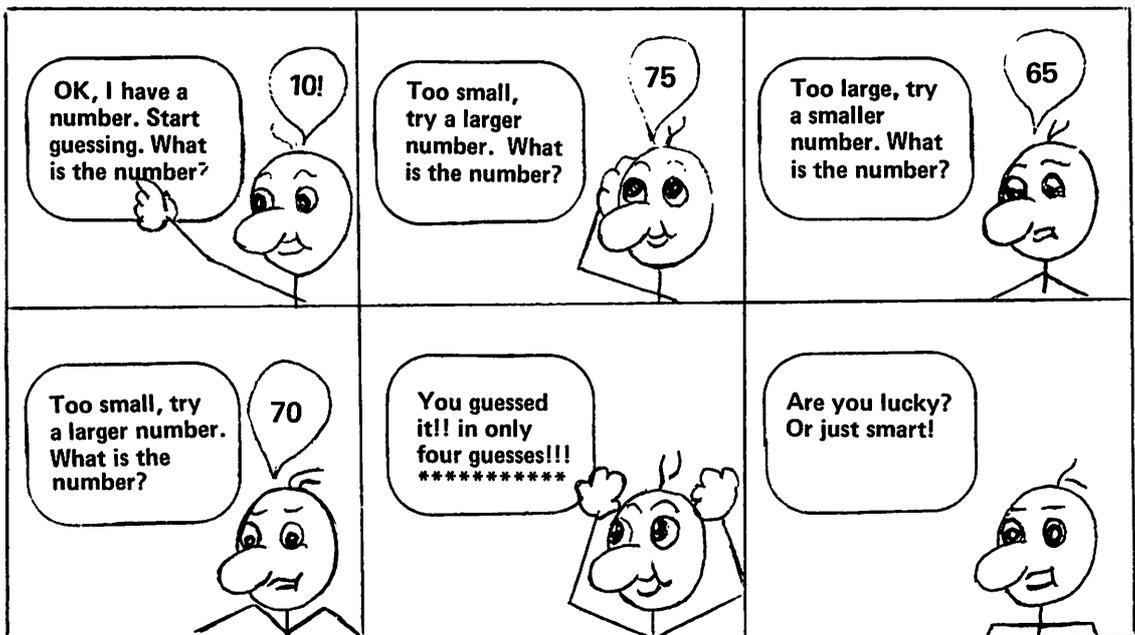
RUN
K= 1
K= 2
K= 3
K= 4
K= 5
K= 6
K= 7
K= 8
K= 9
K= 10
OUT OF LOOP
**DONE**
>□

```

Now the program prints the value of K all the way through 10, because the new line says, "If K is *less than or equal to* 10, branch to line 50."

The IF-THEN statement can be a powerful tool in program development. Experiment now with some programs of your own, using IF-THEN and the mathematical relationships listed on the preceding page.

The IF-THEN statement combined with the RND function provides the opportunity to develop several entertaining simulations on your Home Computer. The programs that follow are based on a number guessing game that many of you may have played. The programs use both the RND function and the IF-THEN statements.



A Number Guessing Program

In this game the computer generates a secret number from 1 to 100, using the RND function, and asks you to guess the number. The program tells you if your guess is larger, smaller, or equal to the secret number. When you guess the number, the program chooses another number and begins the game again. Small children like to play this game with the computer. Big kids like it too!

```

10 REM**NUMBER GUESSING GAME**
20 CALL CLEAR
30 REM**COMPUTER SELECTS A RANDOM NUMBER**
40 SECRET = INT(100*RND)+1 ← Number selected (1 to 100)
50 PRINT "OK, I HAVE A SECRET NUMBER."
60 PRINT
70 INPUT "WHAT IS YOUR GUESS:" : GUESS
80 REM**TEST FOR CORRECT GUESS**
90 IF GUESS = SECRET THEN 180
100 REM**TEST FOR GUESS LARGER THAN NUMBER**
110 IF GUESS > SECRET THEN 150
120 PRINT "TOO SMALL!!" ← If too small
130 PRINT "TRY A LARGER NUMBER."
140 GOTO 60
150 PRINT "TOO BIG!!" ← If too big
160 PRINT "TRY A SMALLER NUMBER."
170 GOTO 60
180 REM**GUESS WAS CORRECT**
190 REM**DISPLAY VICTORY MESSAGE AND PAUSE**
200 PRINT
210 PRINT "YOU GUESSED IT!!" ← If just right
220 PRINT "LET'S PLAY AGAIN"
230 PRINT
240 FOR DELAY = 1 TO 1000 ← Time delay to read victory message
250 NEXT DELAY
260 GOTO 20

```

Two IF-THEN statements are used in the program at lines 90 and 110. In line 90, if the guess is not equal to the secret number, the condition in the IF-THEN statement is false, and the program proceeds to line 100. If the guess is equal to the secret number, the program branches to line 180 and prints the victory message. At line 110, the condition of the guess being larger than the secret number is tested. If the condition is false (the guess is smaller than the number), the program proceeds to line 120. If the condition is true (the guess is larger than the number), a branch to line 150 is made.

Enter the program and run it. See if you can guess the number that the program generates. There is a way of making guesses so that you *always* guess the secret number in *at most seven* trials. Can you discover how to do this?

The following is an example of what might appear on your screen during a RUN of the program.

```

OK, I HAVE A SECRET NUMBER.
WHAT IS YOUR GUESS:35 ← First guess
TOO SMALL!!
TRY A LARGER NUMBER.
WHAT IS YOUR GUESS:75 ← Guess is too big
TOO BIG!!
TRY A SMALLER NUMBER.
WHAT IS YOUR GUESS:50 ← Still too big
TOO BIG!!
TRY A SMALLER NUMBER.
WHAT IS YOUR GUESS:40 ← Now it's too small
TOO SMALL!!
TRY A LARGER NUMBER.
WHAT IS YOUR GUESS:41 ← It must be getting close!
TOO SMALL!!
TRY A LARGER NUMBER.
WHAT IS YOUR GUESS:42 ← Try again
TOO SMALL!!
TRY A LARGER NUMBER.
WHAT IS YOUR GUESS:43 ← Again
TOO SMALL!!
TRY A LARGER NUMBER.
WHAT IS YOUR GUESS:44 ← Again
TOO SMALL!!
TRY A LARGER NUMBER.
WHAT IS YOUR GUESS:45 ← FINALLY DID IT!!!
YOU GUESSED IT!!
LET'S PLAY AGAIN
OK, I HAVE A SECRET NUMBER.

```

The computer will start a new game each time you guess the correct number. When you want to stop playing, just press **SHIFT C**. Notice also that the program does not contain a **RANDOMIZE** statement. Therefore, the program will generate the same series of random numbers each time you run it. If you want to make the program create a new set of random numbers each time, just add this line.

15 RANDOMIZE

A novel version of the number guessing program can be created on your Home Computer using the **SOUND** capabilities of the machine. The next program plays a tone between 131 cycles per second and 247 cycles per second. Your role is to guess the frequency of the tone. (See Appendix) The program lets you know if your guess is lower, higher, or equal to the frequency of the random tone that is generated. When you guess the frequency of the note, the program plays the note three times and begins the game again.

Here is our Tone Guessing Program.

```

10 REM**TONE GUESSING PROGRAM**
20 CALL CLEAR
30 REM**COMPUTER SELECTS A TONE VALUE**
40 TONE = INT(117*RND)+131 ← Selects tone
50 PRINT "OK, I HAVE A TONE"
60 PRINT
70 PRINT "THE TONE IS -"
80 REM**THE TONE IS PLAYED**
90 CALL SOUND(100,TONE,2) ← Play the tone
100 INPUT "WHAT IS YOUR GUESS:" : GUESS
110 REM**TEST FOR CORRECT GUESS**
120 IF GUESS = TONE THEN 250
130 REM**TEST FOR GUESS BEING TOO HIGH**
140 IF GUESS>TONE THEN 200
150 REM**GUESS WAS LOW**
160 CALL SOUND(100,GUESS,2) ← Play the guess
170 PRINT"TOO LOW!!"
180 PRINT "TRY A HIGHER TONE"
190 GO TO 60
200 REM**GUESS WAS HIGH**
210 CALL SOUND(100,GUESS,2) ← Play the guess
220 PRINT "TOO HIGH!!"
230 PRINT "TRY A LOWER TONE"
240 GO TO 60
250 REM**A CORRECT GUESS WAS MADE**
260 REM**PRINT VICTORY MESSAGE AND**
270 REM**PLAY THE SECRET NOTE THREE TIMES**
280 PRINT
290 PRINT "YOU GUESSED IT!!"
300 FOR PLAY = 1 TO 3
310 CALL SOUND(100,TONE,2) ← Play the tone three times
320 NEXT PLAY
330 REM**PAUSE SO MESSAGE CAN BE READ**
340 FOR DELAY = 1 TO 500 ← Delay loop
350 NEXT DELAY
360 GO TO 20

```

Enter the program and RUN it. The information that appears on the screen is similar to the number guessing program. The only difference is that in this program your guess is "played" back to you by the computer. For an interesting variation of this game, remove lines 170–180 and 220–230. These lines contain the visual messages telling you if you are high or low. Without the messages, you have to play the game entirely by "ear."

If you'd like to change the tone limits, you can easily do so by changing line 40. For example, suppose you'd rather hear a series of higher tones — perhaps in the range from 262 cycles per second through 392 cycles per second. How would you rewrite line 40 to generate these tones? Also, you may want to add the RANDOMIZE statement to create a new series of random tones each time that you run the program. If so, just enter this new line:

15 RANDOMIZE

In the last program, if you enter a frequency of less than 110 cycles per second or greater than 5,500 cycles per second, the program stops. An invalid input causes an error condition to occur in the SOUND routine. The IF-THEN statement can be used to detect invalid inputs and is shown in the next program.

This program creates ten randomly placed horizontal bars — of a color you input, and of random lengths. Then the program pauses for you to input a new color code.

You'll notice that we've used IF-THEN statements to test the input color code to be sure it's valid. If it isn't, the program gives you a specially written error message.

```

10 REM**RANDOM COLOR BARS**
20 CALL CLEAR
30 RANDOMIZE
40 REM**INPUT AND VALIDATE COLOR CODE**
50 INPUT "COLOR PLEASE?":C
60 CALL CLEAR
70 IF C<1 THEN 200
80 IF C>16 THEN 200
90 REM**PICK A RANDOM POSITION AND LENGTH**
100 REM**AND DISPLAY 10 BARS**
110 FOR LOOP = 1 TO 10
120 ROW = INT(24*RND)+1
130 SIZE = INT(28*RND)+1
140 CALL COLOR(2,C,C)
150 CALL HCHAR(ROW,3,42,SIZE)
160 FOR DELAY = 1 TO 100
170 NEXT DELAY
180 NEXT LOOP
190 GO TO 230
200 PRINT "INCORRECT COLOR CODE!"
210 PRINT "MUST BE FROM 1 TO 16."
220 PRINT "TRY AGAIN!"
230 FOR DELAY = 1 TO 500
240 NEXT DELAY
250 GO TO 20

```

← Test for valid color code
 ← Generate a row from 1 to 24
 ← Pick size of bar
 ← Display colored bar
 ← Ten bars are on screen — skip down to delay
 ← Special error messages
 ← Give time to view screen
 ← Return to beginning

When you run the program, you'll see all of the bars begin at column 3, near the left-hand edge of the display. Their lengths, however, are random, as are their vertical positions on the screen. After ten bars of the input color are placed, the program clears the screen and asks you for a new color code.

Remember to avoid putting in color codes 1 (transparent) and 4 (the screen color in the Run Mode). Although these are valid codes, you won't be able to see the bars.

The next program is a game where two colors vie against each other for a place on the screen. A winning color is randomly chosen. The program is the longest you've seen yet, so we'll provide some explanations as we go along.

Here's the program:

```

NEW
10**REM COLOR CONTEST**
20 CALL CLEAR
30 REM**ACCEPT AND VALIDATE COLOR CODES**
40 INPUT "FIRST COLOR?":C1
50 IF C1<1 THEN 270
60 IF C1>16 THEN 270
70 INPUT "SECOND COLOR?":C2
80 IF C2<1 THEN 270
90 IF C2>16 THEN 270
100 CALL CLEAR
110 REM**RANDOM PICK OF COLOR**
120 REM**1 = COLOR1 2 = COLOR2**
130 COLORTEST = INT(2*RND)+1
140 REM**DISPLAY 50 RANDOMLY PLACED COLORED SQUARES**
150 FOR LOOP = 1 TO 50
160 ROW = INT(24*RND)+1
170 COLUMN = INT(32*RND)+1
180 IF COLORTEST = 1 THEN 210
190 LET A = C2
200 GO TO 220
210 LET A=C1
220 CALL COLOR(2,A,A)
230 CALL HCHAR(ROW,COLUMN,42)
240 NEXT LOOP
250 GO TO 300
260 REM**ERROR MESSAGES**
270 PRINT "INCORRECT COLOR CODE!"
280 PRINT "MUST BE 1 TO 16."
290 PRINT "TRY AGAIN."
300 REM**WAIT FOR AWHILE**
310 FOR DELAY = 1 TO 500
320 NEXT DELAY
330 GO TO 20

```

Accept first color code and check for validity

Accept second color code and check for validity

This will be 1 or 2

Set loop to display 50 colored squares

Select a random screen position for a square

If Color1 "wins," branch to line 210

If Color2 "wins," assign C2 to A and branch to line 220

If Color 1 "wins," assign value of C1 to A

Print color square in random position

Go through loop until 50 squares are on the screen

Print error message if first color code is invalid

Delay to observe pattern or message; then start new game

Two people can play against each other, or you can play against yourself by putting in both color codes, just to see which "wins" the game. (Again, avoid entering color codes 1 and 4.)

Random Notes

We used `CALL SOUND` earlier in a program that played notes from a musical scale. If we modify that program, adding the `IF-THEN` statement and the `RND` function, we can make the computer play some interesting (but not necessarily enjoyable) "music."

Here's how:

```

NEW
10 REM**COMPUTER MUSIC MAKER**
20 REM**SET UP THE C-SCALE OF NOTES**
30 LET C = 262
40 LET D = 294
50 LET E = 330
60 LET F = 349
70 LET G = 392
80 LET A = 440
90 LET B = 494
100 LET C2 = 523
110 REM**PICK A RANDOM NOTE AND DURATION**
120 RANDOMIZE
130 NOTE = INT(8*RND)+1
140 TIME = INT(100*RND)+100
150 VOLUME = 2
160 REM**IDENTIFY THE NOTE TO BE PLAYED**
170 IF NOTE = 1 THEN 290
180 IF NOTE = 2 THEN 310
190 IF NOTE = 3 THEN 330
200 IF NOTE = 4 THEN 350
210 IF NOTE = 5 THEN 370
220 IF NOTE = 6 THEN 390
230 IF NOTE = 7 THEN 410
240 NOTE = C2
250 REM**THIS LINE PLAYS THE NOTE**
260 CALL SOUND(TIME,NOTE,VOLUME)
270 GO TO 130
280 REM**ASSIGN NOTE TO BE PLAYED**
290 NOTE = C
300 GO TO 260
310 NOTE = D
320 GO TO 260
330 NOTE = E
340 GO TO 260
350 NOTE = F
360 GO TO 260
370 NOTE = G
380 GO TO 260
390 NOTE = A
400 GO TO 260
410 NOTE = B
420 GO TO 260

```

Notes for "middle C" through "high C"

Different random "tune" for each RUN

Set random note and duration

Fix volume at 1

Check which note to be played

Assign "high C" to note

Return for next note

Define the notes

Now run the program, and enjoy the "music." When you're ready to "stop the music," just press **SHIFT C**.

You might like to experiment with this program in various ways. For example, do you notice anything different in the "music" if you change lines 140 and 150 to:

```

140 TIME = 500
150 VOLUME = 5

```

Now that we've let the computer play its own "music," why not play some music of our own?

Musical Interlude

With this program, you can use the keyboard to input notes that you want to play.
Enter these lines:

```

NEW
10 REM**HUMAN ASSISTED COMPUTER MUSIC**
20 REM**SET UP C-SCALE OF NOTES**
30 LET C = 262
40 LET D = 294
50 LET E = 330
60 LET F = 349
70 LET G = 392
80 LET A = 440
90 LET B = 494
100 REM**ACCEPT AND VALIDATE NOTE TO BE PLAYED**
110 INPUT "NOTE ":A$
120 IF A$ = "C" THEN 200
130 IF A$ = "D" THEN 220
140 IF A$ = "E" THEN 240
150 IF A$ = "F" THEN 260
160 IF A$ = "G" THEN 280
170 IF A$ = "A" THEN 300
180 IF A$ = "B" THEN 320
190 GO TO 110
200 NOTE = C
210 GO TO 330
220 NOTE = D
230 GO TO 330
240 NOTE = E
250 GO TO 330
260 NOTE = F
270 GO TO 330
280 NOTE = G
290 GO TO 330
300 NOTE = A
310 GO TO 330
320 NOTE = B
330 REM**PLAY THE NOTE**
340 CALL SOUND(100,NOTE,2)
350 FOR DELAY = 1 TO 50
360 NEXT DELAY
370 GO TO 110

```

"Middle C" scale

Check for which key was depressed on keyboard

Not A through G! Do it again
Set up note to be played

Return for a new note

When you run the program, the computer will ask you for a note. You then press one of the letter keys (A,B,C,D,E, F, or G), followed by the **ENTER** key. For example, when the screen shows

NOTE □

and you press **A ENTER** the "note" A will play. The screen keeps a record of the keys that you depress:

```

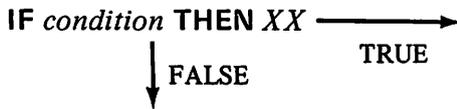
NOTE C
NOTE D
NOTE E
NOTE F

```

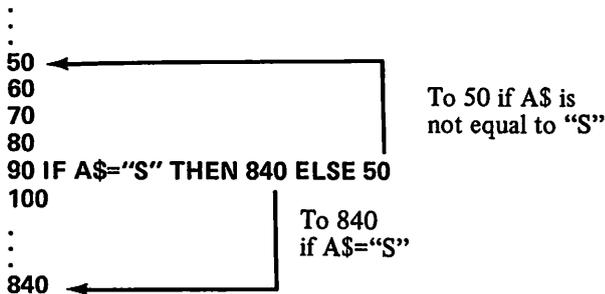
Having to depress the **ENTER** key for each note slows down your musical performance a bit, doesn't it? Also, the **INPUT** statement sounds a note when it is executed. Unless you can work this extra tone into your melody, your music sounds odd. You'll find a quicker and better way to use your computer as an organ keyboard later.

IF-THEN-ELSE Statement

Now that you've become well acquainted with the IF-THEN statement, let's take a look at a variation. Remember the IF-THEN statement allows branching to a nonsequential line number *when the IF condition is true*. Otherwise, it continues on to the next sequential line number.



The IF-THEN-ELSE statement allows branching to one of two different line numbers. It does *not* proceed to the next sequential line unless that line is specified. For example,



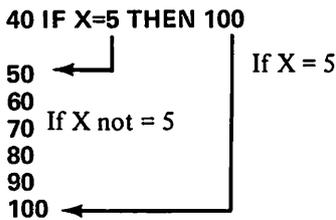
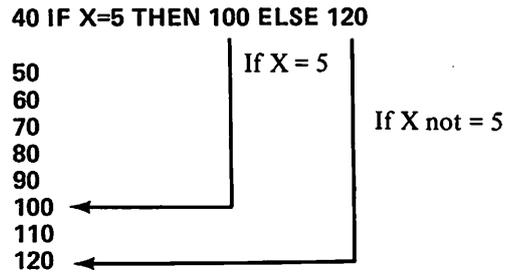
Consider the Musical Interlude program just discussed. Suppose you want to stop the program by typing an S when the program requests a note. You could change line 90 as shown in this example and add line 840.

```

190 IF A$="S" THEN 840 ELSE 110
:
:
840 END
  
```

Now, if no correct note has been pressed, the computer checks A\$ to see if you want to stop or not. If you have typed an S, the computer goes to line 840 and stops. If you have not typed an S, there was an error in your input. The computer goes back to line 110 for a new input.

The difference between IF-THEN and IF-THEN-ELSE is shown in the diagram that follows. When the IF condition is false, the IF-THEN statement always sends you to the next sequential line. This can be changed by using IF-THEN-ELSE where the line number for the false condition may be specified following the word ELSE.

IF-THEN**IF-THEN-ELSE**

The main errors that can occur in using the IF-THEN statement or the IF-THEN-ELSE statement are shown below:

20 IFA=B THEN 200	No space after IF
20 IF A=BTHEN 200 ELSE 300	No space before THEN
20 IF A=B THEN200 ELSE 300	No space after THEN
20 IF A==BTHEN 200	Invalid relational symbol combinations
20 IF A= THEN 200	No expression on one side of the relational symbol

All of the above conditions produce an error message during the running of the program along with a reference to the line number of the statement in which the error occurs.

You have used the GO TO statement in previous programs. When it is executed, control is transferred to the specified line number. Since there are no options, this statement is called an *unconditional* transfer to the *specified* line number.

The ON – GO TO Statement

A more powerful version of the GO TO statement allows a branch to one of several line numbers, depending on a given condition. The ON-GO TO statement is a *conditional* branch to one of several line numbers. The condition, which follows the word ON, may be a numeric expression. This expression is evaluated and rounded to obtain an integer. The value of the integer is then used to select a line number from a list that follows the word GO TO.

Example:

```
30 ON X+1 GO TO 100,150,200
```

- (1) If X+1 evaluates to 1, execution would be transferred to the *first* line number in the list (line 100).
- (2) If X+1 evaluates to 2, execution would be transferred to the *second* line number in the list (line 150).
- (3) If X+1 evaluates to 3, execution would be transferred to the *third* line number in the list (line 200).

(Note: Care must be taken that the condition (X+1) in the example does not evaluate to less than 1 or more than the number of line numbers in the list. If this error condition occurs, the computer stops and displays an error message.)

Look back to the Random Notes program in this chapter. The *seven* IF-THEN statements (lines 170-230) can be replaced by *one* ON-GO TO statement.

70 ON NOTE GO TO 290,310,330,350,370,390,410

NOTE should be equal to 1,2,3,4,5,6 or 7

1st line number 2nd line number 7th line number

Each item in the list following GO TO replaced one of the IF-THEN statements.

IF-THEN In Original	NOTE Value	LIST Item No.	GO TO line
IF NOTE=1 THEN 290	1	1	290
IF NOTE=2 THEN 310	2	2	310
IF NOTE=3 THEN 330	3	3	330
IF NOTE=4 THEN 350	4	4	350
IF NOTE=5 THEN 370	5	5	370
IF NOTE=6 THEN 390	6	6	390
IF NOTE=7 THEN 410	7	7	410

Here is a variation of the Guess My Number program. It uses ON-GO TO in lines 200 and 210.

```

100 REM**GUESS MY NUMBER-VARIATION**
110 CALL CLEAR
120 REM**PICK RANDOMLY 1,2, OR 3**
130 N=INT(3*RND)+1 ← Gives 1,2, or 3
140 PRINT "GUESS MY NUMBER GAME"
150 INPUT "WHAT IS MY NUMBER(1-3)?:":G
160 IF G>3 THEN 140 ← Incorrect input checks
170 IF G<1 THEN 140
180 REM**INPUT OK-NOW CHECK IF CORRECT GUESS**
190 IF N>G THEN 210
200 ON G-N+1 GO TO 240,260,280 ← Compare the two numbers
210 ON N-G GO TO 260,280
220 REM**RESPONSE MESSAGES**
230 REM**CORRECT GUESS**
240 PRINT "RIGHT ON! MY NUMBER WAS";N
250 GO TO 310
260 REM**GUESS OFF BY ONE**
270 PRINT "YOU MISSED IT BY 1.", "MY NUMBER WAS";N
280 REM**GUESS OFF BY TWO**
290 GO TO 310
300 PRINT "YOU MISSED IT BY 2.", "MY NUMBER WAS";N
310 REM**TIME DELAY**
320 FOR W=1 TO 900
330 NEXT W
340 GO TO 110

```

Line 190 is the key to the ON-GO TO statements. If the computer's number is greater than your guess, line 210 is used. In this case, there are two options.

If $N-G=1$, then your message comes from line 260.

If $N-G=2$, then your message comes from line 280.

If the computer's number is less than, or equal to, your guess, line 200 is used. Here there are three options.

If $G-N+1=1$, then your message comes from line 240.

If $G-N+1=2$, then your message comes from line 260.

If $G-N+1=3$, then your message comes from line 280.

Lines 160 and 170 ensure that the rules ($G=1, 2$ or 3) are followed.

The ON – GOSUB Statement

The statement ON-GOSUB has the same relationship to GOSUB as ON-GO TO has to GO TO. GOSUB is an *unconditional* call to a subroutine. ON-GOSUB allows the option of one of several subroutines, depending on the condition following the word ON. For example:

140 ON X GOSUB 500,600,500,600,500,600

X = 1,2,3,4,5 or 6 The same line number may be used more than once

Here is a use of ON-GOSUB in a program.

```

100 REM**ON-GOSUB DEMO**
110 CALL CLEAR
120 FOR X=1 TO 6
130 PRINT "X=";X
140 ON X GOSUB 500,600,500,600,500,600
150 PRINT
160 NEXT X
170 END

500 REM**LONG DELAY**
510 PRINT "TIME DELAY=500"
520 FOR W=1 TO 500
530 NEXT W
540 RETURN

600 REM**SHORT DELAY**
610 PRINT "TIME DELAY=200"
620 FOR W=1 TO 200
630 NEXT W
640 RETURN

```

This program demonstrates the use of ON-GOSUB to select time delays of different lengths. The value of X and the length of the time delay are both printed so that you can follow the execution of the subroutines for each value of X.

Chapter Seven Summary

In this chapter you have learned more ways to alter the normal sequential execution of computer statements. Number guessing and note guessing games have been used to demonstrate these new features. Color Bar demonstrations have also been shown. You have learned to use the following statements:

- The IF-THEN statement, which allows branching depending on the truth of the IF condition.
- ELSE was added to the IF-THEN statement to make it more flexible for branching on the IF condition.
- ON-GO TO provides for branching to one of several program lines depending on the value of the expression following the word ON.
- ON-GOSUB provides for using one of several subroutines depending on the value of the expression following the word ON.

Chapter Seven Exercises

- (1) Given the consecutive statements:

```
50 IF K>10 THEN 180
60 PRINT "THIS IS"; 1979+K
```

If line 50 is executed, will line 60 be executed next if

- (a) $K = 5$? _____
 (b) $K = 10$? _____
 (c) $K = 11$? _____

- (2) Use this part of a program to answer questions a, b and c below.

```
100 CALL CLEAR
110 N = RND
120 IF N<= .33 THEN 170
130 IF N>= .67 THEN 200
140 FOR X = 1 TO 500
150 NEXT X
160 GO TO 220
170 FOR X = 1 TO 50
180 NEXT X
190 GO TO 220
200 FOR X = 1 TO 1000
210 NEXT X
220 PRINT "END OF TIME TEST"
```

- (a) What values of N give the longest time delay? _____
 (b) If $N = .30$, which time delay will be used? _____
 (c) If $N = .33$, which time delay will be used? _____

- (3) Complete this number guessing program. Use the strings: "TOO BIG", "TOO SMALL", and "JUST RIGHT".

```
100 N = INT(100*RND)+1
110 CALL CLEAR
120 INPUT "GUESS =":G
130 IF N>G THEN 170
140 IF N<G THEN 190

150 PRINT " _____ "
160 GO TO 230

170 PRINT " _____ "
180 GO TO 200

190 PRINT " _____ "

200 FOR X = 1 TO 500
210 NEXT X
220 GO TO 110
230 FOR X = 1 TO 1000
240 NEXT X
250 GO TO 100
```

- (4) The color codes for color graphics are the integers 1 to 16. Complete the lines in this partial program so that the correct code range is input. Use IF-THEN statements in lines 40 and 50.

```

20 INPUT "COLOR, PLEASE?":C
30 CALL CLEAR
40 _____
50 _____
:
:
200 PRINT "COLOR CODE INCORRECT!"
210 PRINT "CODE MUST BE 1 TO 16"
220 PRINT "TRY AGAIN."
230 FOR DELAY = 1 TO 500
240 NEXT DELAY
250 GO TO 20

```

- (5) The following lines exist in a program.

```

100 IF A$ = "STOP" THEN 900 ELSE 50
:
:
900 END

```

- (a) If A\$ = "END" and line 100 is executed, what line will be executed next?

- (b) If A\$ is a true statement, which line will be executed next? _____
- (6) Answer the questions below after studying this line.
- ```

70 ON ROLL GO TO 300, 400, 500, 600, 700, 800

```
- (a) If ROLL=3, what line will be executed following line 70? \_\_\_\_\_
- (b) If ROLL=6, what line will be executed following line 70? \_\_\_\_\_
- (c) What value of ROLL would cause line 400 to be executed following line 70?  
\_\_\_\_\_
- (7) Replace the four IF-THEN statements below with one ON-GO TO statement that would accomplish the same result.

```

140 IF NOTE=1 THEN 500
150 IF NOTE=2 THEN 600
160 IF NOTE=3 THEN 700
170 IF NOTE=4 THEN 800

```

140 \_\_\_\_\_

(8) Suppose line 50 below has just been executed.

**50 ON X+2 GOSUB 200,300,400**

(a) If  $X=1$ , which subroutine would be called? \_\_\_\_\_

(b) Is 2 a legal value for  $X$  in this statement? \_\_\_\_\_

(9) Write an ON-GOSUB statement, so that a subroutine at line 400 will be used if  $NOTE < 3$ , line 500 if  $NOTE > 6$  and subroutine 1000 if  $NOTE$  is in the range 3 to 6 inclusive. The legal range for values of  $NOTE$  is 1 to 8.

**200 \_\_\_\_\_**

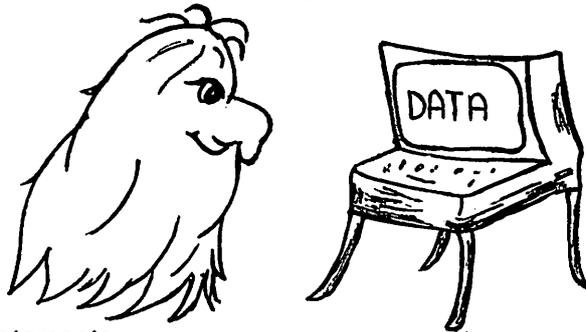
### Answers to Exercises

- (1) (a) yes      (b) yes      (c) no
- (2) (a)  $N$  greater than, or equal to, .67  
 (b) Lines 170–180 (FOR  $X=1$  TO 50, NEXT  $X$ )  
 (c) Lines 140–150 (FOR  $X=1$  TO 1000, NEXT  $Y$ )
- (3) **150 PRINT "JUST RIGHT"**  
**170 PRINT "TOO SMALL"**  
**190 PRINT "TOO BIG"**
- (4) **40 IF  $C < 1$  THEN 200**      (Lines could be in reverse order)  
**50 IF  $C > 16$  THEN 200**
- (5) (a) 50      (b) 900
- (6) (a) 500 (the 3rd item in the list)  
 (b) 800 (the 6th item in the list)  
 (c)  $ROLL=2$  (400 is the 2nd item in the list)
- (7) **140 ON NOTE GO TO 500, 600, 700, 800**
- (8) (a) 400 ( $X+2 = 3$ )  
 (b) no ( $2+2 = 4$  – only 3 items in the list)
- (9) **200 ON NOTE GOSUB 400, 400, 1000, 1000, 1000, 1000, 500, 500**

# Chapter Eight

## Using Data Files

With the addition of more and more programming statements at your command, you are now becoming quite proficient in controlling your computer. You have used assignment statements (with or without the word LET) to assign values to variables. In Chapter Three, you learned to input the values for both numeric and string variables. Now you will explore another method of assignment.



### READ and DATA Statements

Values may be assigned to variables by *reading* them from a *data* list. Every program that contains a READ statement must provide a DATA list to READ from, of course. For example:

```
100 CALL CLEAR
110 READ X ← READ ...
120 PRINT X
130 GO TO 110
140 DATA 1,2,3,4,5,6,7,8 ... from this list
```

The DATA statement is not executed. The items are merely read from it. Therefore, the DATA statement may be placed anywhere in the program. This order would be all right:

```
100 CALL CLEAR
110 READ X
120 DATA 1,2,3,4,5,6,7,8
130 PRINT X
140 GO TO 110
```

If either of the two programs shown were run, you would see:

```

1
2
3
4
5
6
7
8
DATA ERROR IN 110 ← This message just means that
>□ you've used all of your data

```

When your program is entered, the items from the data list are stored sequentially in memory. A data "pointer" is set to "point at" the first item in the data list at the start of a program.

120 DATA 1,2,3,4,5,6,7,8



After the first item of DATA is READ, the pointer moves on to the next item to be ready for the next READ statement.

120 DATA 1,2,3,4,5,6,7,8



This process continues until the last data item in your list has been read. Since there is no more data in the list, the DATA ERROR message is given.

120 DATA 1,2,3,4,5,6,7,8



Pointer here after 8th READ

It would be helpful if you had some way to tell when you reach the end of your data list. Remember the IF-THEN statement? You could insert one at line 125 to avoid the error message.

```

125 IF X=8 THEN 150
150 PRINT "INPUT COMPLETE"

```

Let's see how we can apply this to a practical situation. Remember the CALL COLOR program of Chapter Four? It looked like this:

```

100 CALL CLEAR
110 CALL COLOR(1,6,16)
120 CALL COLOR(2,6,16)
130 CALL COLOR(3,6,16)
140 CALL COLOR(4,6,16)
150 CALL COLOR(5,6,16)
160 CALL COLOR(6,6,16)
170 CALL COLOR(7,6,16)
180 CALL COLOR(8,6,16)
190 PRINT "MY MESSAGE"
200 GO TO 200

```

} 8 CALL COLOR statements

If we use the READ and DATA statements, the program might look like this:

```

100 CALL CLEAR
GO TO 110 READ C
loop to 120 IF C=99 THEN 150
 130 CALL COLOR(C,6,16) ← Only one CALL COLOR statement
READ 140 GO TO 110
data 150 PRINT "MY MESSAGE"
 160 GO TO 160
 170 DATA 1,2,3,4,5,6,7,8,99 ← 99 signals end of list

```

Each time through the loop (lines 110–140), C is tested at line 120. If C is *not* equal to 99, the execution proceeds through the loop. When C=99, we “jump” out of the loop to line 150. Of course, you may substitute any message you wish at line 150.

### More Than One DATA Statement

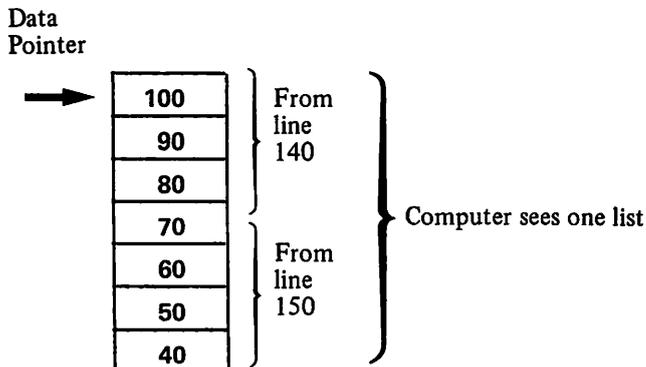
You are not restricted to one DATA statement. If more than one appears in a program, the statements are used in the order in which they occur in the program. When the first one is used up (all items read), items from the next DATA statement are used. In other words, the computer sees them as one extended list. For example:

```

140 DATA 100,90,80 ← These used first
150 DATA 70,60,50,40 ← Then these

```

The computer sees this:



You can make the Home Computer into an expensive adding machine for demonstration purposes. The DATA list could be entries in your check book, with a comfortable starting balance assigned to the variable B. The program will calculate the ending balance. A “flag” will be used to show the last data item in the list.

```

100 REM**CHECK BALANCE PROGRAM**
110 CALL CLEAR
120 REM**B IS THE BEGINNING BALANCE**
130 B=1000
140 REM**READ A CHECK OR DEPOSIT AMOUNT INTO C**
150 READ C
160 REM**LOOK FOR ENDING FLAG -99999**
170 IF C=-99999 THEN 220
180 PRINT C
190 REM**ADJUST BALANCE BY CHECK/DEPOSIT AMOUNT**
200 B= B+C
210 GO TO 150
220 PRINT "ENDING BALANCE=";B
230 DATA -17.41,-82.56,-5.40,-72.13
240 DATA -175,650,47,-350,-45.92
250 DATA -55.01,-73.50,-99999

```

← Flag to tell when end  
of data is reached

Trace the values for C and B as the GO TO loop is executed.

| PASS NO. | C | NEW B |
|----------|---|-------|
| 0        | - | 1000  |
| 1        |   |       |
| 2        |   |       |
| 3        |   |       |
| 4        |   |       |
| 5        |   |       |
| 6        |   |       |
| 7        |   |       |
| 8        |   |       |
| 9        |   |       |
| 10       |   |       |
| 11       |   |       |

← Starting balance

← Ending balance

(The results are shown on the next page.)

Remember the Random Notes program in Chapter Seven? With READ, DATA and ON-GO TO statements we can now shorten that program.

```

100 REM**RANDOM NOTES**
110 REM**INPUT NOTES**
120 READ C,D,E,F,G,A,B,C2
130 REM**SET UP THE NOTE TO BE PLAYED**
140 RANDOMIZE
150 NOTE=INT(8*RND)+1
160 TIME=INT(1000*RND)+100
170 VOLUME=2
180 REM**CHECK WHAT NOTE WAS SELECTED**
190 ON NOTE GO TO 260,280,300,320,340,360,380,210
200 REM**HIGH C WAS CHOSEN**
210 NOTE=C2
220 REM**PLAY THE NOTE**
230 CALL SOUND(TIME,NOTE, VOLUME)
240 GO TO 150
250 REM**SET NOTE TO APPROPRIATE VALUE**
260 NOTE=C
270 GO TO 230
280 NOTE=D
290 GO TO 230
300 NOTE=E
310 GO TO 230
320 NOTE=F
330 GO TO 230
340 NOTE=G
350 GO TO 230
360 NOTE=A
370 GO TO 230
380 NOTE=B
390 GO TO 230
400 DATA 262,294,330,394
410 DATA 392,440,494,523

```

(Results from preceding page)

| PASS NO. | C       | NEW B   |
|----------|---------|---------|
| 0        | —       | 1000    |
| 1        | -17.41  | 982.59  |
| 2        | -82.56  | 900.03  |
| 3        | -5.40   | 894.63  |
| 4        | -72.13  | 822.50  |
| 5        | -175.00 | 647.50  |
| 6        | 650.47  | 1297.97 |
| 7        | -350.00 | 947.97  |
| 8        | -45.92  | 902.05  |
| 9        | -55.01  | 847.04  |
| 10       | -73.50  | 773.54  |
| 11       | -99999  | 773.54  |

Recall the Musical Scale Program in Chapter Four? We used a long list of assignment statements for the notes, middle C, D, E, F, G, A, B and high C. To play the notes, we had to use many BASIC statements. The program below shows how a DATA list and a READ statement shortens the program. If we use a FOR-NEXT loop, we can eliminate the need for a flag at the end of the DATA list.

```

100 REM**MODIFIED MUSICAL SCALE PROGRAM**
110 REM** T IS DURATION;L IS LOUDNESS**
120 T=100
130 L=2
140 REM**LOOP OVER ALL NOTES**
150 FOR X=1 TO 15
160 READ N
170 CALL SOUND(T,N,L)
180 NEXT X
190 END
200 DATA 262,294,330,349,392
210 DATA 440,494,523,494,440
220 DATA 392,349,330,294,262

```

The original program was 25 lines long. So we've cut it to less than half of what it was. However, if you run this new program it will go up and down the scale only once. The previous program ran continuously up and down. If we could use the data list over and over again, we could make this program work the same way. BASIC is an amazing language. Every time we need a new instruction (or statement), there seems to be one that satisfies our need. Such is the case now.

### The RESTORE Statement

When the computer executes a RESTORE statement, the data pointer moves back to the beginning of your DATA list. You can now use the DATA all over again.

In our Musical Scale program, we change line 190 and add line 195 as follows:

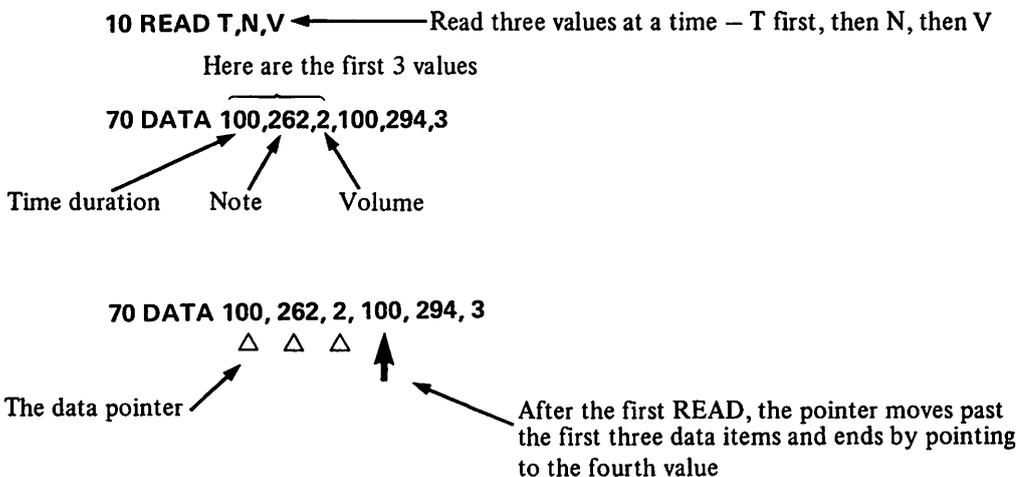
```
190 RESTORE
195 GO TO 150
```

Now when the program is run, we hear the same continuous scale that we heard in Chapter Four. Our new program is only 14 lines long.

You can now substitute any data that you want in the DATA list and play "songs" of your own choosing. The program that you now have plays all the notes for the same time duration and at the same volume. These can be custom designed to fit your needs by reading in the duration and volume for each note just as you read the note in this program.

### READ a List

READ statements may contain more than one variable. A whole list of variables can be read. In our Custom Fitted Notes Program, below, we'll READ the time duration, the volume, and the note itself.



```

100 REM**CUSTOM FITTED NOTES PROGRAM**
110 REM**LOOP TO READ AND PLAY NOTES**
120 FOR X=1 TO 8
130 READ T,N,V
140 CALL SOUND(T,N,V)
150 NEXT X
160 RESTORE Data pointer is moved to beginning of list here
170 GO TO 120
180 DATA 100,262,2,100,294,3
190 DATA 200,330,2,100,349,3
200 DATA 100,392,2,100,440,3
210 DATA 100,494,2,500,523,2

```

The program plays up the scale with varying note durations and volumes. By changing only the DATA items, you may customize your own music. If you add more notes, the upper limit of the FOR statement in line 120 must be changed accordingly.

### String DATA

DATA statements may contain strings as well as numeric values, but the type of DATA (string or numeric) must correspond to the type of variable used in the READ statement. Numeric variables require numeric constants, and string variables require quoted, or unquoted, strings. For example,

```

120 READ N$, K

```

String   ↕   ↘  
          ↕   ↘  
180 DATA "CHECK",-17.41

or

```

210 DATA "DEPOSIT",650.47

```

Going back to the checkbook program that we used earlier in the chapter, we could label each entry by making these program modifications:

```

100 REM**CHECK BALANCE PROGRAM**
110 CALL CLEAR
120 REM**B IS THE BEGINNING BALANCE**
130 B=1000
140 REM**READ LABEL AND AMOUNT**
150 READ N$,C
160 REM**LOOK FOR ENDING FLAG -99999**
170 IF C=-99999 THEN 220
180 PRINT N$,C
190 REM**ADJUST BALANCE BY CHECK/DEPOSIT AMOUNT**
200 B=B+C
210 GO TO 150
220 PRINT "ENDING BALANCE=";B
230 DATA "CHECK",-17.41
240 DATA "CHECK",-82.56
250 DATA "CHECK",-175
260 DATA "DEPOSIT",650.47
270 DATA "END",-99999

```

(Note: You must provide a string to be read with this flag, -99999, since the READ statement asks for a pair of inputs.)

The printout of the results now provide a clearer picture of the transactions that have taken place. Checks and deposits will be labeled. The quotation marks in the DATA lists are optional but are used here to emphasize that these are strings assigned to a string variable.

### Plain and Fancy PRINTing

When you used the PRINT statement in the Immediate Mode, you saw the difference in spacing between printed numeric values using a comma or a semicolon as a separator. Let's take another look at this. Try each of the following examples. In each, assume that the screen has been cleared by typing CALL CLEAR and pressing ENTER.

Type PRINT 1,2 and press ENTER.

```
>PRINT 1,2
 1 2
>□
```

Type PRINT 1,2,3 and press ENTER.

```
>PRINT 1,2,3
 1 2
 3
>□
```

Type PRINT 1,2,3,4,5,6,7 and press ENTER.

```
>PRINT 1,2,3,4,5,6,7
 1 2
 3 4
 5 6
 7
>□
```

As you have probably guessed, with comma spacing the computer will print up to two items on each line. If the PRINT statement has more than two items, the computer simply continues on the next line until all items have been printed.

Now let's look at semicolon spacing.

Type **PRINT 1;2** and press **ENTER**.

```
>PRINT 1;2
 1 2
>□
```

Type **PRINT 1;2;3** and press **ENTER**.

```
>PRINT 1;2;3
 1 2 3
>□
```

Type **PRINT 1;2;3;4;5;6;7** and press **ENTER**.

```
>PRINT 1;2;3;4;5;6;7
 1 2 3 4 5 6 7
>□
```

So far, you have used only small positive integers. Try some negative numbers.

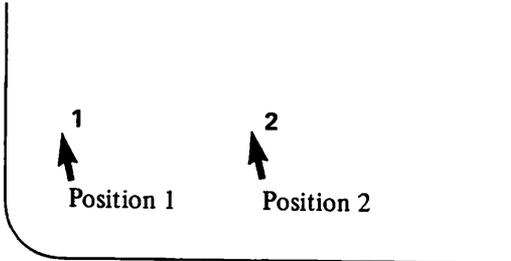
Type **PRINT -1;-2** and press **ENTER**.

```
>PRINT -1;-2
-1 -2
>□
```

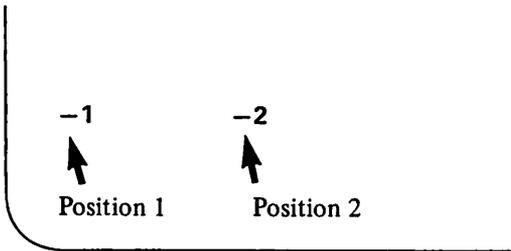
Type **PRINT -1;-2;-3;-4** and press **ENTER**.

```
>PRINT -1;-2;-3;-4
-1 -2 -3 -4
>□
```

Now let's discuss the rules for comma and semicolon spacing. With comma spacing, items are printed in *standard printing positions*. There are two such positions on the screen. Position 1 is at the extremem left. Position 2 is halfway across the screen.



Note that the computer always leaves a space preceding the number for the "sign" of the number. For positives, the plus sign (+) is assumed and is not printed on the screen. For negatives, the computer prints a minus sign (-) before the number.



Now look at the semicolon spacing.

Type `PRINT 1;22;333` and press `ENTER`.

```
>PRINT 1;22;333
 1 22 333
>□
```

Type `PRINT -1;-22;-333` and press `ENTER`.

```
>PRINT -1;-22;-333
 -1 -22 -333
>□
```

The semicolon instructs the computer *not to leave any spaces* between the values or variables in the `PRINT` statement. Then why do we see spaces between the numbers on the screen? There are two reasons.

- (1) Remember that each positive number is preceded by a space for its sign.
- (2) Every number is followed by a *trailing space*. The trailing space is there to guarantee a space between all numbers, even negative numbers.

Now, instead of short, simple numbers, let's try some long, messy numbers.

### Comma Spacing

Type **PRINT 4/3,8/3** and press **ENTER**.

```
> PRINT 4/3,8/3
 1.333333333 2.666666667
> □
```

Notice that the semicolon causes the second number to be moved left one additional space.

### Semicolon Spacing

Type **PRINT 4/3;8/3** and press **ENTER**.

```
> PRINT 4/3;8/3
 1.333333333 2.666666667
> □
```

Let's try some examples with string variables, using commas as separators.

```
> LET A$="ZONE 1"
> LET B$="ZONE 2"
> PRINT A$,B$
 ZONE 1 ZONE 2
> □
```

The "strings" (the letters and numbers within quotation marks) are also printed in different zones on the screen when a comma is used to separate the string variables. Try this set of statements.

```
> LET A$="ONE"
> LET B$="TWO"
> LET C$="THREE"
> LET D$="FOUR"
> PRINT A$,B$,C$,D$
 ONE TWO
 THREE FOUR
> □
```

If the semicolon tells the computer to leave no spaces between variables in a PRINT statement, what happens when we use string variables rather than numeric?

```
>LET A$="HI THERE!"
>LET B$="HOW ARE YOU?"
>PRINT A$;B$
 HI THERE!HOW ARE YOU?
>□
```

↙ They ran together – no space!

As you saw, the two strings had no space between them. If we want a space to appear between them, we must include the space inside one of the sets of quotation marks. For example, change A\$:

```
LET A$="HI THERE! "
PRINT A$;B$
```

↙ One space

```
>LET A$="HI THERE!"
>LET B$="HOW ARE YOU?"
 PRINT A$;B$
 HI THERE!HOW ARE YOU? } First example
>LET A$="HI THERE! "
>PRINT A$;B$
 HI THERE! HOW ARE YOU? } Second example
>□
```

There is a third separator that can be used – the colon. The colon instructs the computer to print the next item at the beginning of the next line. It works the same way with both numeric and string variables. Enter these lines as an example:

```
LET A=-5
LET B$="HELLO"
LET C$="MY NAME IS ALPHA"
PRINT A:B$:C$
```

```
>LET A=-5
>LET B$="HELLO"
>LET C$="MY NAME IS ALPHA"
>PRINT A:B$:C$
-5
HELLO
MY NAME IS ALPHA
>□
```

To review for a moment, these are the three print separators that we have used.

| <b>Punctuation Mark</b> | <b>Operation</b>                                                          |
|-------------------------|---------------------------------------------------------------------------|
| Comma                   | Prints values in different print zones;<br>maximum of two items per line. |
| Semicolon               | Leaves no spaces between items (except<br>those mentioned for numerics).  |
| Colon                   | Prints next item on the following line.                                   |

### The TAB Function

Another method of controlling the format of a PRINT statement is through the TAB function. This function is similar to that of a typewriter TAB key. You may state how many spaces to indent the printed matter by means of the TAB function.

Example: TAB(10) would indent 10 spaces from the left margin. Therefore, the statement:

```
PRINT TAB(10);"THREE"
```

would cause the word THREE to be printed with the letter T in the tenth space in from the left margin.

```
>PRINT TAB(10);"THREE"
 THREE
>□
```

Notice that the print line on the screen has 28 columns or character positions. Thus the first position on the print line counts as column 1. This is where the P appears in the word "PRINT" on the screen above. The last print position on the line is column 28.

You can also use the TAB function more than once in a PRINT statement.

```
>PRINT TAB(10);3;TAB(20);-4
 3 -4
>□
```

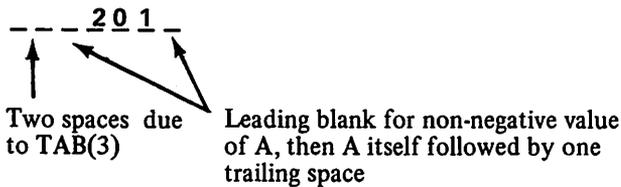
Notice that the first number (3) is actually printed in column 11 because the preceding or "leading" space (reserved for the sign of the number) occupies column 10, just as the minus sign of the second number (-4) occupies column 20.

The TAB function always starts counting in column 1 (the leftmost print position on the line), regardless of where or how many times it appears in the PRINT statement. In the example above, the second number (-4) was printed starting in the twentieth column on the print line, *not* 20 spaces from the position at which the first number (3) was printed.

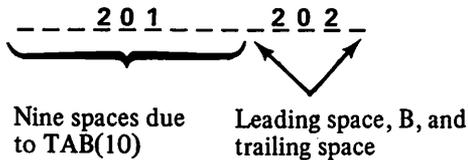
Here is another example for analysis that involves multiple TABs.

```
LET A=201
LET B=202
LET C=203
CALL CLEAR
PRINT TAB(3);A;TAB(10);B;TAB(17);C
```

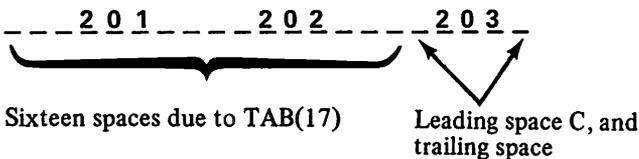
(1) TAB(3);A; causes



(2) TAB(10);B; causes



(3) TAB(17);C; causes



When all of the TABs are used together in a single PRINT statement the result looks like this:

```

201 202 203
>□
```

What happens if we indicate a column that is already occupied by another message, or if there isn't enough room left on the line to print the message positioned by a TAB? Enter the following short program to find the answer.

```
NEW
10 CALL CLEAR
20 LET A$="HELLO! HOW ARE YOU?"
30 LET B$="HI!"
40 PRINT A$;TAB(5);B$
50 PRINT B$;TAB(20);A$
60 END
```

Now RUN the program.

```
> □
HELLO! HOW ARE YOU?
 HI!
HI!
HELLO! HOW ARE YOU?
```

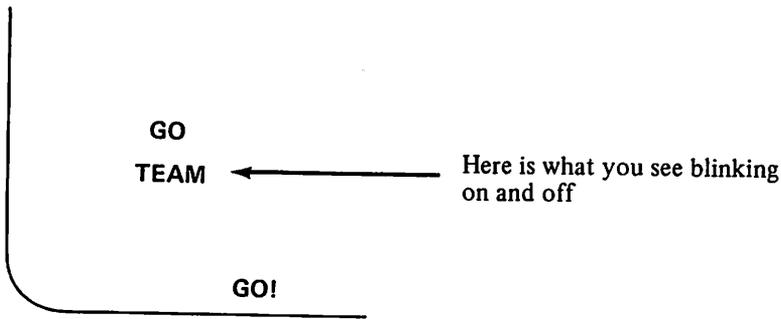
Line 40: column 5 is already taken by the O in HELLO. So HI! starts in column 5 of the next line

Line 50: the whole message of A\$ won't fit on a line if it starts at column 20, so "HELLO! HOW ARE YOU?" is printed starting in column 1 of the next line

Now, imagine that your Home Computer is connected to a TV with a giant screen. It is the big game of the season, and your computer is in front of the home team fans, running the following program. Try it!

```
100 REM**CHEER LEADING PROGRAM**
110 CALL CLEAR
120 REM**LOOP TO PUT UP CHEER THREE TIMES**
130 FOR X = 1 TO 3
140 REM**DISPLAY MESSAGE ELEMENTS**
150 PRINT TAB(13);"GO"
160 PRINT ← Prints a blank line
170 PRINT TAB(12);"TEAM"
180 PRINT
190 PRINT TAB(13);"GO!"
200 REM**DELAY TO SEE MESSAGE**
210 FOR Z = 1 TO 1000
220 NEXT Z ← Time delays
230 CALL CLEAR
240 REM**DELAY TO KEEP SCREEN BLANK**
250 FOR Z=1 TO 500
260 NEXT Z
270 NEXT X
```

Type NEW, enter the above program and RUN it. The computer will blink the message GO TEAM GO! on the screen three times.



Note that GO TEAM GO! is approximately centered left to right. This was done with the TAB function in lines 150, 170 and 190. If you wish to move the message up toward the center of the screen, put several empty PRINT statements between lines 190 and 200. Or you can use the following FOR-NEXT loop.

```
192 FOR UP=1 TO 9
194 PRINT
196 NEXT UP
```

Presto! GO TEAM GO! is now center stage...er, center screen that is. May the best team win!

The two programs we will develop next continue our exploration of computer graphics by showing how to construct patterns out of standard characters. Although the statements and functions used in the programs are elements of BASIC that you already know, you may see some new applications of these features.

## Rectangles and Squares

The first program allows you to place a rectangle or square of standard characters on the screen. In the previous chapters of this book, you've identified characters by their *character codes*, numbers 32 through 96. (The characters and their corresponding codes are listed in the Appendix.) Now, we want to demonstrate a new technique: assigning a character to a string variable from the keyboard.

Try these examples in the Immediate Mode before entering the program.

```
LET A$=""
PRINT A$
PRINT A$;A$
PRINT A$;TAB(10);A$
```

```

>LET A$="*"
>PRINT A$
* ← One asterisk

>PRINT A$;A$
** ← Two asterisks, side by side

>PRINT A$;TAB(10);A$
* * ← One asterisk at left margin, one in column 10

>□

```

Try a few more Immediate Mode experiments on your own. For example, what would happen if you redefined A\$ as "\*\*\*\*" or as "(")? Try it and see what happens!

This method is convenient if you want to print only a short line of characters. But what if you want to print long lines or vary the line length or character that the program prints? INPUT statements and a FOR-NEXT loop will solve the problem. Type NEW, then enter this program.

```

100 REM**PRINTING CHARACTER PATTERNS**
110 CALL CLEAR
120 REM**GET CHARACTER AND SIZE OF PATTERN**
130 INPUT "CHARACTER?":A$
140 INPUT "WIDTH?":W
150 REM**LOOP TO PRINT ROW OF CHARACTERS**
160 FOR X=1 TO W
170 PRINT A$;
180 NEXT X

```

When you run the program, you'll first be asked to input the character that you want to use. Just type the character and press ENTER. Then you'll be asked for the width, or the number of characters in the line that you want to print. Type in the number and press ENTER to continue the program. Suppose you entered \* as the character and 28 as the width. The screen would look something like this:

```

CHARACTER?*
WIDTH?28

DONE

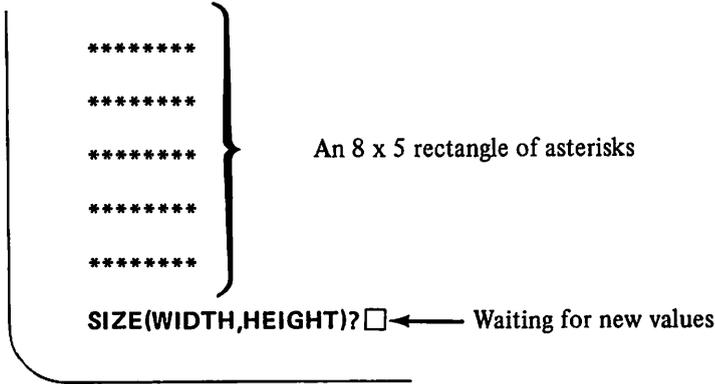
>□

```

RUN the program a few times, entering different characters and lengths. Then try adding the following lines to allow you to make rectangles and squares of characters.

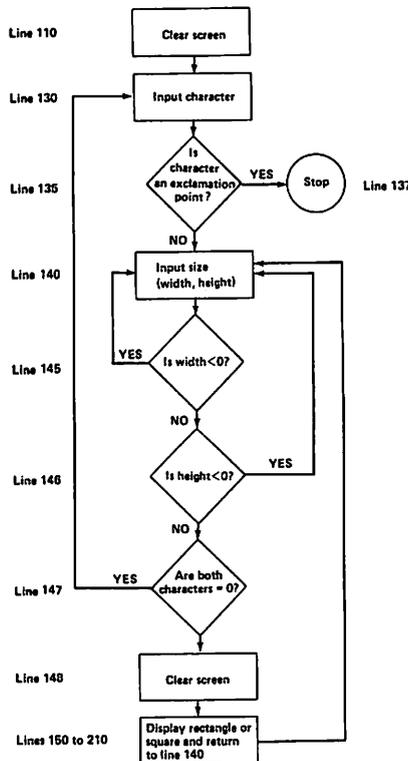


RUN the program. For this example, enter \* when the program asks CHARACTER? Then enter 8,5 when you're asked for width and height:



Experiment with the program. Try entering the control values (width and height both zero) so that you can change the character. Vary width and height so that the display fills the screen or makes only tall, thin bars and wide, flat strips. What happens if you enter a width greater than 28 or a height greater than 12? Try it and see what happens.

Perhaps a flow chart will help to describe how the program works. The following diagram doesn't show the whole program in detail; it covers only the parts that relate to program control by input values.



**“Holes”**

Let's expand the Rectangles and Squares program one more time. These new lines will create rectangles or squares with a random sprinkling of “holes” (blank spaces) in the display field. Enter the following lines:

```

115 RANDOMIZE
162 IF INT(2*RND)=0 THEN 170
164 PRINT " ";
166 GO TO 180

```

Now clear the screen and list the changed program so that we can discuss the effect of these additions.

```

LIST
100 REM**PRINTING CHARACTER PATTERNS**
110 CALL CLEAR
115 RANDOMIZE
120 REM**GET CHARACTER AND SIZE OF PATTERN**
130 INPUT "CHARACTER?":A$
135 IF A$ "!" THEN 140
137 END
140 INPUT "SIZE(WIDTH,HEIGHT)?":W,H
145 IF W<0 THEN 140
146 IF H<0 THEN 140
147 IF W+H=0 THEN 130
148 CALL CLEAR
150 FOR Y= 1 TO H
160 FOR X= 1 TO W
162 IF INT(2*RND)=0 THEN 170
164 PRINT " ";
166 GO TO 180
170 PRINT A$;
180 NEXT X
190 PRINT : :
200 NEXT Y
210 GO TO 140

```

The test with the RND function in line 162 causes a character to be printed whenever INT(2\*RND) is equal to 0; a space when INT(2\*RND) is equal to 1. Approximately half the time the program prints a character and half the time a space. RUN the program now and observe the kind of pattern that emerges.

**Chapter Summary**

In this chapter you have learned some new tools for assigning string and numeric variables. You have also learned to control an exit from a GO TO loop. You have used this new technique in sound and color programs. You can now INPUT data and output screen displays in more and better ways. You have learned how:

- To READ data from a DATA list
- To avoid OUT OF DATA errors
- A data pointer works
- To use an IF-THEN statement with a flag
- To READ in data for use in creating sound and color
- To READ data for use in arithmetic applications
- To use more than one DATA statement in a program
- To trace the steps of the computer as it READS and uses data
- To READ more than one item in a READ statement
- To use commas, semicolons, and colons as item separators
- To use the TAB function for precise spacing

## Chapter Eight Exercises

- (1) Complete this statement.

If a program contains a READ statement, it must also include a \_\_\_\_\_ statement.

- (2) If a READ statement is executed 5 times, and there are only 4 items in the program's only DATA statement, what happens (assume the RESTORE statement has
- not*
- been used)?

- (3) If
- $X = 15$
- and line 190 (below) is executed, what line will be executed next? \_\_\_\_\_

```
190 IF X>15 THEN 300
200 GO TO 100
```

- (4) Study this program. Then answer the questions that follow.

```
100 CALL CLEAR
110 READ C
120 IF C=555 THEN 150
130 PRINT C
140 GO TO 110
150 END
160 DATA 500,495,520,555,313
```

- (a) How many values of C would be printed? \_\_\_\_\_
- (b) How many values of C would be read? \_\_\_\_\_
- (c) Would the data item 313 be read? \_\_\_\_\_
- (d) Would the data item 555 be printed? \_\_\_\_\_

- (5) Complete this statement.

If two DATA statements are used in a program, the computer reads the two as

\_\_\_\_\_.

(one extended list, two separate lists)

- (6) Suppose the two lines below were the only READ and DATA statements in a program.

```
10 READ T,N,V
100 DATA 100,262,2,200,294,2
```

- (a) When line 10 is executed the first time, what value is assigned to N? \_\_\_\_\_
- (b) What value would be assigned to N the second time that line 10 is executed?
- \_\_\_\_\_

- (7) If these are the only DATA lines in a program, show the order in which the computer stores (or holds) the data items.

```
180 DATA "CHECK",-43.10,"CHECK"
190 DATA -17.45,"DEPOSIT",355.55
```

Order: 1 \_\_\_\_\_  
 2 \_\_\_\_\_  
 3 \_\_\_\_\_  
 4 \_\_\_\_\_  
 5 \_\_\_\_\_  
 6 \_\_\_\_\_

- (8) Complete.
- (a) For wide spacing, separate the items in a PRINT statement by \_\_\_\_\_.
- (b) For close spacing, separate the items in a PRINT statement by \_\_\_\_\_.
- (c) For printing on separate lines, separate items in a PRINT statement by \_\_\_\_\_.
- (9) If a semicolon is used to separate string variables in a PRINT statement, describe how the strings are displayed.
- 

- (10) How can you use strings to print words with spaces between the words?
- 

- (11) This statement is executed: `PRINT TAB(5);"FIVE"`  
 At what space on the printed display would the V of "FIVE" appear? \_\_\_\_\_

- (12) Show what would be printed and where the results are displayed when this program is RUN.

```
10 A=131
20 B = -313
30 C=345
40 CALL CLEAR
50 PRINT TAB(2);A;TAB(11);B;TAB(19);C
```

-----  
 5            10            15            20            25

**Answers to Exercises**

- (1) DATA statement
- (2) The computer gives an OUT OF DATA ERROR message.
- (3) Line 200 (then line 100 – X=15 but *is not* greater than 15)
- (4) (a) 3 (500,495 and 520)  
 (b) 4 (500, 495, 520 and 555)  
 (c) no (when C=555 the program stops)  
 (d) no (line 120 causes execution to go to line 150 – line 130 is skipped)
- (5) As one extended list
- (6) (a) 262  
 (b) 294
- (7) Order: 1 CHECK  
 2 -43.10  
 3 CHECK  
 4 -17.45  
 5 DEPOSIT  
 6 355.55
- (8) (a) commas  
 (b) Semicolons  
 (c) colons
- (9) They run together with no spacing in between.
- (10) Include spaces in the string (inside the quotes).
- (11) 7th space from the left (the word FIVE would start 5 spaces in).

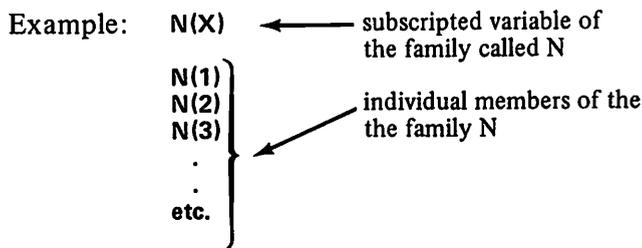
(12)   1  3  1     -  3  1  3     3  4  5       
           5          10          15          20          25

# Chapter Nine

## One Dimensional Arrays

In Chapter Eight you learned to use data that were entered with READ, DATA, and RESTORE statements. In this chapter, we will introduce subscripted variables and one-dimensional arrays.

Subscripted variables are similar to the simple variables that you have been using for numeric values and strings, except that subscripted variables are followed by a number in parentheses. They can be thought of as a family of variables, and the number in parentheses distinguishes which member of the family is being referred to.



One-dimensional arrays are simply ordered lists with each data item assigned a unique number by its subscript. When we read in items from a DATA statement, the items were READ in the order that they appeared in the DATA list. Items from an array may be used in any order since each item is numbered and may be accessed by that number. Once used, the item remains in the array, and no RESTORE statement is required to use it again.

In the Modified Musical Scale program (Chapter Eight), we read in the note values 262,294, 330,349,392,440,494, and 523. We also read the notes in the reverse order. We will now show how to assign these values to the subscripted variable:



The variable names the array. The subscript (number in parentheses) labels one element of that array. Thus we can assign these values:

|            |                |
|------------|----------------|
| N(1) = 262 | (first note)   |
| N(2) = 294 | (second note)  |
| N(3) = 330 | (third note)   |
| N(4) = 349 | (fourth note)  |
| N(5) = 392 | (fifth note)   |
| N(6) = 440 | (sixth note)   |
| N(7) = 494 | (seventh note) |
| N(8) = 523 | (eighth note)  |

Now, when you want to play a particular note (say 440), you could use the statement:

**SOUND(1000,N(6),2)**

duration                      note                      volume

The size of an array is limited by the amount of memory that is in your computer. Memory space used by a one-dimensional array is normally set for eleven array elements having the subscripts 0 through 10. However, you may increase the memory space reserved for an array by using a DIMension statement.

If we wanted to place 20 notes in our array, we would reserved space with the statement:

**DIM N(20)**

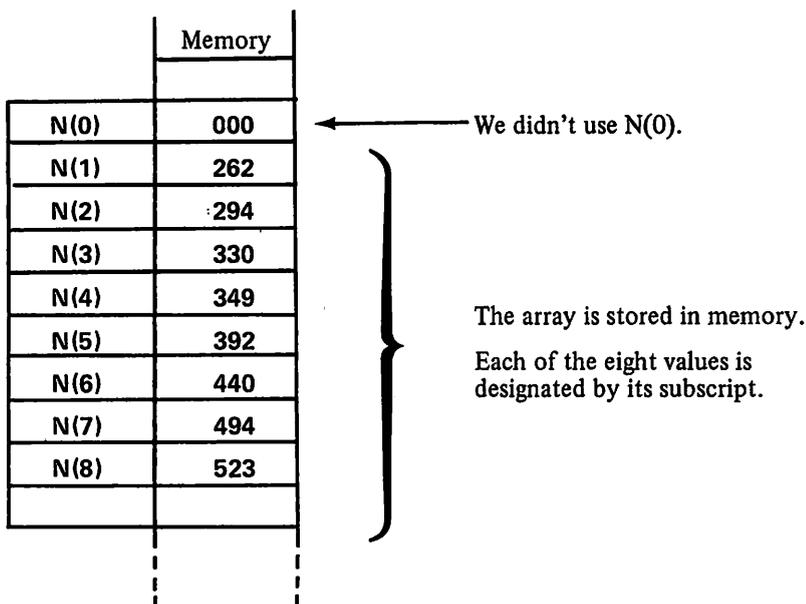
dimension                      array N                      for 20 notes  
(actually 21, N(0) through N(20))

An array can be dimensioned only *once* in a given program. The dimension statement must occur *before* the array is used in the program. If no dimension statement is given, memory space for the array is limited so that it holds only 11 elements.

The data placed in the array may be either numbers or strings depending on how the array is defined. DIM N(20) would define an array consisting of 21 numeric values. DIM N\$(20) would define an array consisting of 21 alphanumeric strings.

Although the zero subscripted variable is available, it does not have to be used. Most people like to start their array with the number one.

It is convenient to think of a one-dimensional array as a series of items stored in separate memory "boxes." The eight notes used in the previous example are held in memory in the order of their subscripts.



You can assign the values to an array through the READ and DATA statements used in the last chapter. Since our array has only eight elements, we do not need a DIMENSION statement.

Example:

```

100 CALL CLEAR
110 FOR X = 1 TO 8 ← Loop to read 8 values
120 READ N(X) ← and assign them to N(1)
130 NEXT X ← through N(8).

500 DATA 262,294,330,349
510 DATA 392,440,494,523

```

When the above lines are executed, the array is stored in the computer. Any note may now be taken from the array and played. Try playing some random notes by adding the following lines to those above, making a complete program.

```

200 RANDOMIZE
210 FOR S = 1 TO 30 ← Play 30 notes
220 Y = INT(8*RND)+1 ← Select a random value (1 through 8)
230 CALL SOUND(500,N(Y),2) ← duration, note, loudness
240 NEXT S

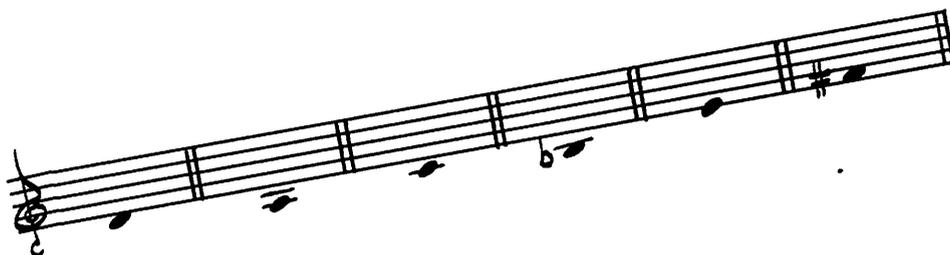
```

Some variations to the program can be provided by changing the upper limit in line 210 to play more or fewer notes. You may also want to vary the duration and loudness. If you do,

```

add 225 T = INT(901*RND)+100 ← duration 100 through
 1000 milliseconds
add 227 V = INT(31*RND) ← volume 0 through 30
and change 230 CALL SOUND(T,N(Y),V)

```



### Tone Guessing Game

Test your ear for tones with the Tone Guessing Game which follows. The computer will pick a random tone (C,D,E,F,G,A or B) in the scale of C. You will then be asked to pick the number of the correct tone. The values you must input are shown in this table with the appropriate matching tone.

| INPUT | TONE |
|-------|------|
| 1     | C    |
| 2     | D    |
| 3     | E    |
| 4     | F    |
| 5     | G    |
| 6     | A    |
| 7     | B    |

If your guess is incorrect, the computer will tell you if your guess was too high or too low. It will then play the note again and ask for another input. If you are correct, the computer confirms your guess with an appropriate message, including the note's frequency. The program uses two arrays – N and N\$. The N array contains the frequency of the notes; N\$ the name of each note.

## TONE GUESSING GAME

```

100 CALL CLEAR
110 REM*DIRECTIONS*
120 PRINT "WHEN YOU HEAR THE SOUND,"
130 PRINT "TYPE THE NUMBER OF THE"
140 PRINT "NOTE THAT YOU THINK"
150 PRINT "WAS PLAYED."

200 REM*READ IN TONES AND NOTE NAMES*
210 FOR X = 1 TO 7
220 READ N(X),N$(X)
230 NEXT X

300 REM*PICK RANDOM TONE*
310 Y=INT(7*RND)+1

400 REM*PLAY AND COMPARE*
410 CALL SOUND(500,N(Y),2)
420 INPUT "TONE NUMBER?":T
430 IF T>Y GO TO 600
440 IF T<Y GO TO 700
450 PRINT "THAT'S IT!"
460 PRINT "MY TONE WAS";N(Y)
470 PRINT "CYCLES PER SECOND WHICH"
480 PRINT "IS THE NOTE";
490 PRINT N$(Y); " IN THE SCALE OF C."

500 PRINT
510 GO TO 310

600 REM* HIGH ERROR *
610 PRINT "TOO HIGH! TRY LOWER."
620 GO TO 720

700 REM*LOW ERROR*
710 PRINT "TOO LOW! TRY HIGHER."
720 FOR W = 1 TO 200
730 NEXT W
740 GO TO 400

800 DATA 262,C,294,D,330,E,349,F
810 DATA 392,G,440,A,494,B

```

Remember, you must input the note *number*, *not* the letter of the note.

## Color Organ

The Tone Guessing Game used two single one-dimensional arrays to hold the notes in the C-scale. Next, let's use two arrays to build a color organ. One will hold color values, and the other will hold note values. We'll use them to combine colors on the screen with sounds from the speaker. This will give us a simple color organ with notes chosen in a random fashion. Since the program will choose the appropriate color for the note selected, the

sounds will be color coordinated. A low-frequency color will be displayed with a low-frequency sound; high frequency colors will be displayed with high-frequency sounds. The colors will be displayed near the center of the screen.

First the data is read into the arrays.

```

100 CALL CLEAR

200 REM* READ DATA TO ARRAYS *
210 FOR X = 1 TO 8
220 READ N(X),C(X) N(X)=tone, C(X)=color
230 NEXT X

```

Now we come to the heart of the program, where a random value from 1 through 8 is selected and the color associated with this note is placed on the screen. The note is then played. The color stays on for a short time after the note finishes, then a new note is selected, and the sequence repeats. Twenty notes will be played before the program ends.

```

300 REM* COLOR AND SOUND *
310 FOR T = 1 TO 20
320 Y = INT(8*RND)+1
330 CALL COLOR(2,C(Y),C(Y))
340 CALL HCHAR(16,10,42,10)
350 CALL SOUND(200,N(Y),2)
360 FOR W = 1 TO 100
370 NEXT W
380 NEXT T

```

We also need to add the DATA list for line 220 to READ. Notice that the order of the data is: NOTE 1, COLOR 1, NOTE 2, COLOR 2, etc. . . .

```

500 DATA 262,7,294,9,330,11,349,12
510 DATA 392,13,440,6,494,5,523,14

```

Here are the tables showing the values used in the two arrays.

Sounds

| Array Element | Freq. | Note   |
|---------------|-------|--------|
| N(1)          | 262   | C      |
| N(2)          | 292   | D      |
| N(3)          | 330   | E      |
| N(4)          | 349   | F      |
| N(5)          | 392   | G      |
| N(6)          | 440   | A      |
| N(7)          | 494   | B      |
| N(8)          | 523   | High C |

Colors

| Array Element | Color Code | Color        |
|---------------|------------|--------------|
| C(1)          | 7          | Dark red     |
| C(2)          | 9          | Medium red   |
| C(3)          | 11         | Dark Yellow  |
| C(4)          | 12         | Light Yellow |
| C(5)          | 13         | Dark green   |
| C(6)          | 6          | Light blue   |
| C(7)          | 5          | Dark blue    |
| C(8)          | 14         | Magenta      |

## Customizing the Color Organ

With one minor change to the Color Organ program, you can choose and play your own notes. Line 320 randomly picked one of the eight notes. If you change that line as follows, you will be able to input one of the eight notes each time the program goes through the FOR-NEXT loop.

```
320 INPUT "NOTE :1 THROUGH 8?":Y
322 CALL CLEAR
```

Let's also move the color swatch around the screen depending on whether you play a high or low note. This can be done by altering the CALL HCHAR statement in line 340. We'll also add lines 333 and 336 to calculate the parameters for the CALL HCHAR statement.

```
333 ROW = 9 - Y ← row
336 COL = Y + 2 ← column
340 CALL HCHAR(ROW, COL, 42, 5)
```

Now the color swatch is not so long, and it will move about in the following way as the different notes are played.

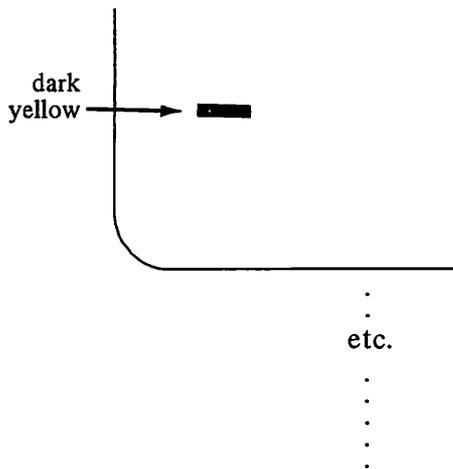
If INPUT = 1, then row = 8 and column = 3



If INPUT = 2, then row = 7 and column = 4

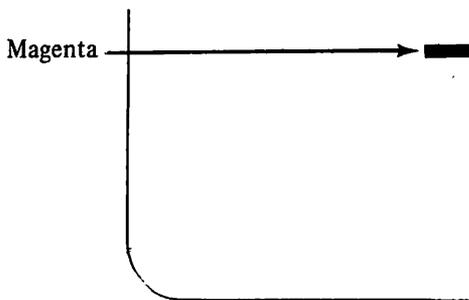


If INPUT = 3, then row = 6 and column = 5



next higher note = E

If INPUT = 8, then row = 1 and column = 10



highest note = high C

Type LIST on your computer, and you will see the Customized Color Organ program.

```

100 CALL CLEAR
200 REM* READ DATA TO ARRAYS *
210 FOR X = 1 TO 8
220 READ N(X),C(X)
230 NEXT X
300 REM* COLOR AND SOUND *
310 FOR T = 1 TO 20
320 INPUT "NOTE:1 THROUGH 8?":Y
322 CALL CLEAR
330 CALL COLOR(2,C(Y),C(Y))
333 ROW=9-Y
336 COL=Y+2
340 CALL HCHAR(ROW,COL,42,5)
350 CALL SOUND(200,N(Y),2)
360 FOR W = 1 TO 100
370 NEXT W
380 NEXT T
500 DATA 262,7,294,9,330,11,349,12
510 DATA 392,13,440,6,494,5,523,14

```

### Other Uses for Arrays

Arrays are useful wherever data must be manipulated in a nonsequential way. Suppose a store owner wishes to keep track of weekly sales. At the end of each week, salesmen turn in their weekly sales reports. The data is entered in the computer and totalled.

For example, a store might want the itemized totals for magazine, book, computer, power supply, keyboard, and video display sales. The total number of items sold might also be desired.

An array could be used to hold the data as follows:

**S(1)**= magazines

**S(2)**= books

**S(3)**= computers

**S(4)**= power supplies

**S(5)**= keyboards

**S(6)**= video displays

↙ **S(0)**= Total of S(1) through S(6)  
 We finally make use of  
 the zero position in an array

The steps in the program are:

- (1) Clear the screen and set each item in the array to zero.

```
100 CALL CLEAR
110 FOR N = 0 TO 6
120 S(N) = 0
130 NEXT N
```

- (2) Input the data and keep a running sum. When all entries have been made, INPUT 99 for the item number.

```
200 INPUT "ITEM NUMBER(1 THROUGH 6)?:":N
210 IF N = 99 THEN 300
220 INPUT "HOW MANY?":A
230 S(N) = S(N) + A
240 GO TO 200
```

- (3) Total all the categories.

```
300 FOR Z = 1 TO 6
310 S(0) = S(0) + S(Z)
320 NEXT Z
```

(4) Print results.

```

400 PRINT "MAGAZINES =";S(1)
410 PRINT "BOOKS =";S(2)
420 PRINT "COMPUTERS =";S(3)
430 PRINT "POWER SUPPLIES =";S(4)
440 PRINT "KEYBOARDS =";S(5)
450 PRINT "VIDEOS =";S(6)
460 PRINT "TOTAL SALES =";S(0)

```

Enter the program in the computer. The store has five salesmen who turn in the following sales totals. Run the program using this data. Input each salesman's totals separately.

| ITEM           | SALESMEN |       |      |      |       |
|----------------|----------|-------|------|------|-------|
|                | CANE     | BAKER | ABLE | DEAL | HOLMS |
| Magazines      | 14       | 7     | 12   | 22   | 17    |
| Books          | 5        | 8     | 4    | 12   | 9     |
| Computers      | 2        | 1     | 0    | 0    | 3     |
| Power Supplies | 0        | 3     | 0    | 1    | 1     |
| Keyboards      | 1        | 4     | 1    | 0    | 0     |
| Videos         | 1        | 0     | 2    | 0    | 3     |

Sample INPUT for Salesman Cane

```

.
.
.
ITEM NUMBER(1 THROUGH 6)?1
HOW MANY?14
ITEM NUMBER(1 THROUGH 6)?2
HOW MANY?5
ITEM NUMBER(1 THROUGH 6)?3
HOW MANY?2
ITEM NUMBER(1 THROUGH 6)?5 ← Cane had no sales from
 item 4
HOW MANY?1
ITEM NUMBER(1 THROUGH 6)?6
HOW MANY?1
ITEM NUMBER(1 THROUGH 6)?□ ← Now input Baker's sales for items 1 through 6

```

When all the salesmen's entries have been made, the totals are shown on the screen.

```

HOW MANY?3
ITEM NUMBER(1 THROUGH 6)?99 ← Signals the end of data
MAGAZINES = 72
BOOKS = 38
COMPUTERS = 6
POWER SUPPLIES = 5
KEYBOARDS = 6
VIDEOS = 6
TOTAL SALES = 133

** DONE **
>□

```

You might want to make an addition to the program which would show the total sales in dollars. If all the magazines were identically priced, all the book prices were identical, etc., you would merely multiply each item total by the unit price and sum the totals.

### Calculating Salesmen's Commissions

Our computer store has decided to calculate and print weekly sales records of its salesmen. The individual reports will show how many of each item was sold and the salesman's commission on that item as well as his total commission.

Arrays used in the program:

|         |                                   |
|---------|-----------------------------------|
| A(N)    | number of each item sold by Able  |
| B(N)    | number of each item sold by Baker |
| C(N)    | number of each item sold by Cane  |
| D(N)    | number of each item sold by Deal  |
| H(N)    | number of each item sold by Holms |
| ACOM(N) | commission on each item for Able  |
| BCOM(N) | commission on each item for Baker |
| CCOM(N) | commission on each item for Cane  |
| DCOM(N) | commission on each item for Deal  |
| HCOM(N) | commission on each item for Holms |
| N\$(N)  | names of each item                |

Variables used in the program:

|     |                            |
|-----|----------------------------|
| TA  | total commission for Able  |
| TB  | total commission for Baker |
| TC  | total commission for Cane  |
| TD  | total commission for Deal  |
| TH  | total commission for Holms |
| S\$ | salesman's name            |
| N   | item number                |
| S   | number of items sold       |
| P   | commission for each item   |
| X,Y | loop counters              |

Values used in the program:

|                    |         |
|--------------------|---------|
| Magazine price     | \$2     |
| Book price         | \$9.95  |
| Computer price     | \$1000  |
| Commission         | 40%     |
| Power Supply price | \$49.50 |
| Keyboard price     | \$125   |
| Video price        | \$150   |

### COMMISSION CALCULATOR

```

100 CALL CLEAR
110 REM** CLEAR TOTALS **
120 TA = 0
130 TB = 0
140 TC = 0
150 TD = 0
160 TH = 0
170 REM** INPUT NAMES OF PRODUCTS **
175 FOR X = 1 TO 6
180 READ N$(X)
185 NEXT X

200 INPUT "SALESMAN?":S$
210 IF S$ = "DONE" THEN 800
220 INPUT "ITEM NUMBER?":N
230 IF N = 99 THEN 200
240 INPUT "HOW MANY?":S
250 IF S$ = "CANE" THEN 400
260 IF S$ = "BAKER" THEN 500
270 IF S$ = "ABLE" THEN 600
280 IF S$ = "DEAL" THEN 700
290 IF S$ <> "HOLMS" THEN 200

300 H(N) = S
310 ON N GOSUB 1000,1020,1040,1060,1080,1100
320 HCOM(N) = P
330 TH = TH+P
340 GO TO 220

400 C(N) = S
410 ON N GOSUB 1000,1020,1040,1060,1080,1100
420 CCOM(N) = P
430 TC = TC+P
440 GO TO 220

500 B(N) = S
510 ON N GOSUB 1000,1020,1040,1060,1080,1100
520 BCOM(N) = P
530 TB = TB+P
540 GO TO 220

600 A(N) = S
610 ON N GOSUB 1000,1020,1040,1060,1080,1100
620 ACOM(N) = P
630 TA = TA+P
640 GO TO 220

```

Initialize totals

Salesman's last name —  
type done if finished

Type 99 if salesman's items  
have all been entered

```
700 D(N) = S
710 ON N GOSUB 1000,1020,1040,1060,1080,1100
720 DCOM(N) = P
730 TD = TD+P
740 GO TO 220

800 PRINT "CANE COMMISSION REPORT"
805 FOR Y = 1 TO 6
810 PRINT C(Y);N$(Y);" = ";CCOM(Y)
815 NEXT Y
820 PRINT "TOTAL COMMISSION";TC
825 PRINT
830 PRINT "BAKER COMMISSION REPORT"
835 FOR Y = 1 TO 6
840 PRINT B(Y);N$(Y);" = ";BCOM(Y)
845 NEXT Y
850 PRINT "TOTAL COMMISSION";TB
855 PRINT
860 PRINT "ABLE COMMISSION REPORT"
865 FOR Y = 1 TO 6
870 PRINT A(Y);N$(Y);" = ";ACOM(Y)
875 NEXT Y
880 PRINT "TOTAL COMMISSION";TA
885 PRINT
890 PRINT "DEAL COMMISSION REPORT"
895 FOR Y = 1 TO 6
900 PRINT D(Y);N$(Y);" = ";DCOM(Y)
905 NEXT Y
910 PRINT "TOTAL COMMISSION";TD
915 PRINT
920 PRINT "HOLMS COMMISSION REPORT"
925 FOR Y = 1 TO 6
930 PRINT H(Y);N$(Y);" = ";HCOM(Y)
935 NEXT Y
940 PRINT "TOTAL COMMISSION";TH
950 END

1000 P=S*2*.4
1010 RETURN

1020 P=S*9.95*.4
1030 RETURN

1040 P=S*1000*.4
1050 RETURN

1060 P=S*49.50*.4
1070 RETURN

1080 P=S*125*.4
1090 RETURN

1100 P=S*150*.4
1110 RETURN

2000 DATA "MAGAZINES","BOOKS","COMPUTERS"
2010 DATA "POWER SUPPLIES","KEYBOARDS","VIDEOS"
```

### How To Run the Program

After the program has been entered, run it using the data from the previous program. This program will clear the screen, initialize the variables used for the running sums of the salesmen's total commissions, and read in the item name array. The computer will then ask for the salesman's name.

```
SALESMAN?□
```

After you type in the name, it will ask for the item number.

```
SALESMAN?CANE
ITEM NUMBER?□
```

When the item number has been entered, it will ask how many of that item has been sold (by that salesman).

```
SALESMAN?CANE
ITEM NUMBER?1
HOW MANY?□
```

After the number sold has been entered, the computer will select the appropriate line to execute (400,500,600,700, or 300) depending on the current salesman being processed (400 in the case of Cane). The number of sales is entered for the appropriate item in the appropriate array.

For Cane:  $C(1) = 14$  ← 14 magazines (item 1) sold by Cane

The commission is then calculated in the subroutine selected by the item number.

For Cane:  $N = 1$  GOSUB 1000  
 $P = 14 * 2 * .40 = 11.2$   
 14 mags.      \$2 each      40% commission

P is then that salesman's (Cane's) commission on that item (magazines)

On return from the subroutine, the salesman's commission for that item is entered in the appropriate array.

For Cane:  $CCOM(1) = 11.2$

His running total is then calculated.

For Cane:  $TC = TC + P$   
 $TC = 0 + 11.2 = 11.2$

The program then returns to line 220 for the next item number.

At this point, the screen shows only:

```
SALESMAN?CANE
ITEM NUMBER?1
HOW MANY?14
ITEM NUMBER?□
```

The process above is repeated for each item sold by that salesman. When all of a salesman's items have been entered, type 99 for the item number. The computer will then ask for the next salesman's name.

```

SALESMAN?CANE
ITEM NUMBER?1
HOW MANY?14
ITEM NUMBER?2
HOW MANY?5
ITEM NUMBER?3
HOW MANY?2
ITEM NUMBER?5 ← Cane sold no power supplies
HOW MANY?1
ITEM NUMBER?6
HOW MANY?1
ITEM NUMBER?99
SALESMAN?□

```

When all entries have been made for all salesmen, type DONE in answer to the request for the salesman's name. The computer will then print the report for each salesman.

A typical report:

```

CANE COMMISSION REPORT
14 MAGAZINES = 11.2
5 BOOKS = 19.9
2 COMPUTERS = 800
0 POWER SUPPLIES = 0
1 KEYBOARDS = 50
1 VIDEOS = 60
TOTAL COMMISSION 941.1

```

**Chapter Nine Exercises**

(1) Locate the subscripted variables from those shown below:

- |          |           |
|----------|-----------|
| (a) TA   | (d) \$\$5 |
| (b) T(3) | (e) XY(3) |
| (c) T3   | (f) TA3   |

- (2) Is a RESTORE STATEMENT necessary in order to use an item of an array more than once? \_\_\_\_\_
- (3) May an array have more than 11 items? \_\_\_\_\_
- (4) Write a dimension statement that will provide for 25 elements in an array using the variable B. \_\_\_\_\_
- (5) How many times may an array be dimensioned in a given program?  
\_\_\_\_\_

Questions 6 through 8 use the information in this array.

|           |           |            |
|-----------|-----------|------------|
| A(0) = 0  | A(5) = 16 | A(10) = 50 |
| A(1) = 10 | A(6) = 13 | A(11) = 40 |
| A(2) = 20 | A(7) = 30 | A(12) = 22 |
| A(3) = 25 | A(8) = 14 |            |
| A(4) = 7  | A(9) = 35 |            |

- (6) Write a dimension statement for the array above. \_\_\_\_\_
- (7) What is the value of A(5) + A(8)? \_\_\_\_\_
- (8) If the above array had been stored in the computer, and the following lines of a program were run, what value will be stored in A(0)?

```
100 P = A(11) + A(2)
110 A(0) = P + A(4)
```

A(0) = \_\_\_\_\_

(9) Suppose you are running the Tone Guessing Game on page 166. The computer has picked the tone of 330 cycles per second (the note E). You have input the number 4 as your guess. What message will appear on the screen?

\_\_\_\_\_

\_\_\_\_\_

- (10) Suppose that you are running the Color Organ program on page 167. The value selected at random for Y in line 320 was 4.
- a. What color swatch will be displayed? \_\_\_\_\_
  - b. What tone will sound? \_\_\_\_\_
- (11) a. Which salesman earned the most commission according to the Commission Calculator program?  
\_\_\_\_\_
- b. How much commission did he earn? \_\_\_\_\_

**Answers to Chapter Nine Exercises**

- (1) The variables at (b), (d), and (f) are subscripted variables.
- (2) No
- (3) Yes (if it is properly DIMensioned)
- (4) DIM B(24) (We'll also accept DIM B(25))
- (5) only once
- (6) DIM A(12)
- (7) 30 (16+14)
- (8) 67 (P = 40+20 = 60, A(0) = 60+7 = 67)
- (9)

TOO HIGH! TRY LOWER.  
TONE NUMBER?□

- (10) a. Light yellow
- b. F (349)
- (11) a. Holms
- b. \$1419.22

# Chapter Ten

## Two Dimensions and Beyond

In the last chapter, you were introduced to the use of one-dimensional arrays on your TI Home Computer. You found that using arrays and the DIMENSION statement helped you save programming effort. You were able to construct compact routines that accomplished quite a lot.

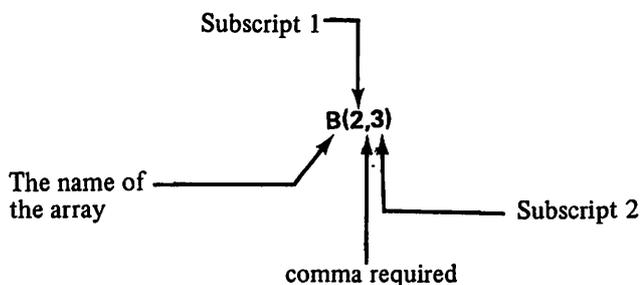
In this chapter we'll examine multi-dimensional arrays. We will begin with arrays of two dimensions, and then progress to those of three dimensions. On the TI Home Computer, *three-dimensional* arrays are the largest dimensioned arrays that you can use.

We will show you how to store and retrieve information from multi-dimensional arrays and how to use arrays effectively to reduce your programming efforts even more, and we'll also touch on the use of multi-dimensional string arrays.

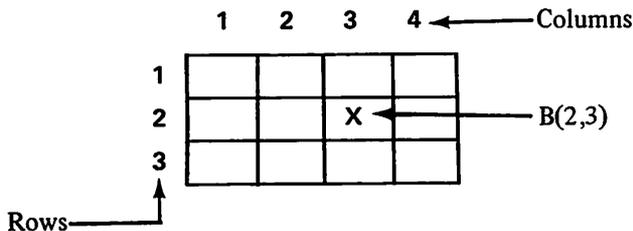
Of course, we will use the arrays to look once again at some of the familiar programs of sound, color, and graphics that you have seen before. To start, let's see how a two-dimensional array is constructed in TI BASIC.

For one-dimensional arrays or lists, as they are sometimes called, we indicated that the variable was a dimensioned array by the use of a subscript. What do you think we use to indicate two-dimensional arrays? Correct! We use *two* subscripts.

If we have a two-dimensional array whose dimensions are of size 3 x 4 and whose label (or name) is B, an element of that array might be:



If the array is thought of as a table, the first subscript indicates the row of the table where the element is located. The second subscript indicates the column of the table. So, for our example, the element  $B(2,3)$  is located at row 2, column 3 of the array B.



How many elements are in the array B? That's right! There are 3 times 4 elements, or 12 locations in the array that can hold values. If the array B were filled with values, it might look something like this:

|   |    |    |   |    |           |
|---|----|----|---|----|-----------|
|   | 1  | 2  | 3 | 4  | ← Columns |
| 1 | 12 | 3  | 0 | 7  |           |
| 2 | -9 | 1  | 5 | 4  |           |
| 3 | 6  | 14 | 8 | 10 |           |

Rows ↑

Now we can talk about the specific value of  $B(2,3)$ . Look at the table that represents the array. What is the value of  $B(2,3)$ ? Correct!  $B(2,3)$  has a value of 5. The location in the TI computer's memory called  $B(2,3)$  contains the value 5. Similarly, the array element  $B(1,4)$  has the value 7.

If the computer had stored the above values in the array B, and we executed the statements:

```
PRINT B(2,3)
PRINT B(1,4)
```

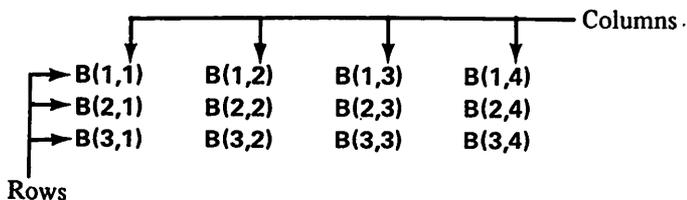
We would see:

```

5
7

```

as the output. If we look at the 'name' of each element in the B array, we would find the following:



The last diagram gives you a picture of what the B array elements *look* like. This information especially the ordering of the data elements, is important to know, when READ-DATA statements are used to fill the arrays. We will have more to say on that later in the chapter. For now, just notice how the array elements are labeled.

You may be wondering how the values that are shown on the last page got into the B array. Well, there are several ways that it could have been done. The most direct way is by using the assignment statements. For example, the following would put the value 5 in B(2,3):

**B(2,3) = 5**

If this were done in the Immediate Mode, the line shown above would do the job. If the assignment were needed inside a program, a line number in front of the assignment would be sufficient:

**10 B(2,3) = 5**

So multi-dimensioned array elements behave just like simple variables in terms of doing things with them inside a program. You can add them together, do any kind of arithmetic on them, print them on the screen, and put the value of any array element into another variable location by using the assignment statement. All of the following operations are legal in TI BASIC.

|                                |   |                                  |
|--------------------------------|---|----------------------------------|
| 10 B(2,3) = 5                  | } | Assignment statements            |
| 20 B(1,2) = 3                  |   |                                  |
| 30 B(3,3) = B(2,3) + B(1,2)    | } | Arithmetic operations            |
| 40 B(2,4) = B(2,3) - B(2,2)    |   |                                  |
| 50 B(2,1) = -(B(2,2) + B(3,3)) |   |                                  |
| 60 PRINT B(3,2)                |   | ← Printing on screen             |
| 70 Z = B(3,4)                  |   | ← Assignment to another variable |

Before we go into specific uses of the two-dimensional array feature, let's look at how the array we have been discussing could be filled with the use of the READ and DATA statements. In order to do this, we must use a set of nested FOR-NEXT loops. Type NEW to clear your machine, and enter the following small program:

```

10 REM**READ AND PRINT ARRAY**
20 FOR ROW = 1 TO 3
30 FOR COL = 1 TO 4
40 REM**READ AND PRINT A ROW**
50 READ B(ROW,COL)
60 PRINT B(ROW,COL);
70 NEXT COL
80 PRINT
90 REM**GET NEXT ROW**
100 NEXT ROW
110 DATA 12,3,0,7,-9,1,5,4,6,14,8,10

```

If you RUN the program, the values of the DATA statement will be transferred into the appropriate locations of the B array. The elements will also appear on the screen. Notice that if you interchange the order of the FOR-NEXT loops, the DATA statement elements must be rearranged as well. For example, the following program performs the same task of filling the B array as the routine listed above, look at how the DATA statement must be entered:

```

10 REM**READ AND PRINT ARRAY**
20 FOR COL = 1 TO 4
30 FOR ROW = 1 TO 3
40 REM**READ BY COLUMNS**
50 READ B(ROW,COL)
60 NEXT ROW
70 NEXT COL
80 DATA 12,-9,6,3,1,14,0,5,8,7,4,10

```

In both of the examples shown above, the B array, at the end of program execution, contains exactly the same values as shown below:

B(ROW,COL)

|   |    |    |   |    |
|---|----|----|---|----|
| R | 12 | 3  | 0 | 7  |
| O | -9 | 1  | 5 | 4  |
| W | 6  | 14 | 8 | 10 |
| S |    |    |   |    |

↑      ↑      ↑      ↑  
COLUMNS

Are you beginning to understand how the multi-dimensional variables work? Great! Let's see how we might use the two-dimensional feature in a program. We'll put together a program that plays notes on the musical scale while displaying colors on the screen — a sort of "color organ."

To make this program, we want to use the C-scale that you have heard before and associate with each note a particular color. The table given below shows what we will need:

| NOTE | FREQUENCY | COLOR CODE | COLOR        |
|------|-----------|------------|--------------|
| C    | 262       | 5          | Dark blue    |
| D    | 294       | 7          | Dark red     |
| E    | 330       | 11         | Dark yellow  |
| F    | 349       | 14         | Magenta      |
| G    | 392       | 6          | Light blue   |
| A    | 440       | 8          | Cyan         |
| B    | 494       | 10         | Light red    |
| C    | 523       | 12         | Light yellow |

First, let's build the part of the program that reads the data into a two-dimensional array. We will use the array name B, and will store the frequency in the first column, the color in the second. The array dimensions are 8 rows by 2 columns. Type NEW and enter the following partial program:

```

10 FOR ROW = 1 TO 8
20 REM**LOAD FREQUENCY COLUMN 1**
30 FOR COL = 1 TO 2
40 READ B(ROW,COL) ← Pairs of frequencies and colors
50 REM**LOAD COLOR-COLUMN 2**
60 NEXT COL
70 NEXT ROW
80 DATA 262,5,294,7,330,11,349,14
90 DATA 392,6,440,8,494,10,523,12

```

This part of the program loads the B array with pairs of data for the note to be played and the color to be shown. The array looks like this when this section is run:

|             |   |                   |    |             |   |
|-------------|---|-------------------|----|-------------|---|
|             |   | <u>B(ROW,COL)</u> |    |             |   |
|             |   | 262               | 5  |             |   |
|             |   | 294               | 7  |             |   |
|             |   | 330               | 11 |             |   |
| Frequencies | → | 349               | 14 |             |   |
|             |   | 392               | 6  |             |   |
|             |   | 440               | 8  |             |   |
|             |   | 494               | 10 |             |   |
|             |   | 523               | 12 |             |   |
| Rows        | ↑ |                   |    | Color codes | } |
|             |   | 1                 | 2  |             |   |
|             |   | Columns           |    |             |   |

Now, we add the rest of the program that plays the notes and displays the colors. The colors will appear in the middle of the screen and will form a small square that changes color with the notes being played. You may enter and run the following:

```

95 REM**SET PARAMETERS**
100 T = 500 ← Set duration (T) and
110 V = 2 ← loudness (V)
120 REM**GET FREQ AND COLOR**
130 FOR ROW = 1 TO 8
140 TONE = B(ROW,1) ← Get frequency
150 C = B(ROW,2)
160 REM**DISPLAY SQUARE**
170 CALL COLOR(2,C,C) ← Set foreground and background
180 CALL HCHAR(12,14,42,2) to same color
190 CALL HCHAR(13,14,42,2)
200 REM**SOUND ON**
210 CALL SOUND(T,TONE,V) ←
220 NEXT ROW
230 GO TO 130 ← Go back and do it again

```

This part of the program sets the duration (T) and loudness (V) for the SOUND routine. A loop is set up to cycle over the eight notes (line 130). The frequency of the tone to be played is taken from the array B (line 140). The color code is assigned from B (line 150). Line 170 calls the COLOR routine. The foreground and background colors are set the same. This action will produce a spot of color on the screen when a character is displayed. At line 180 and line 190 two color spots are displayed on the screen. The result is a small square, two units on a side, near the center of the monitor. The SOUND routine is then called for the current note. The loop handles all eight notes and colors, and then (line 230) the program branches back to line 130 to play the notes again. The program continues until you press **SHIFT C**.

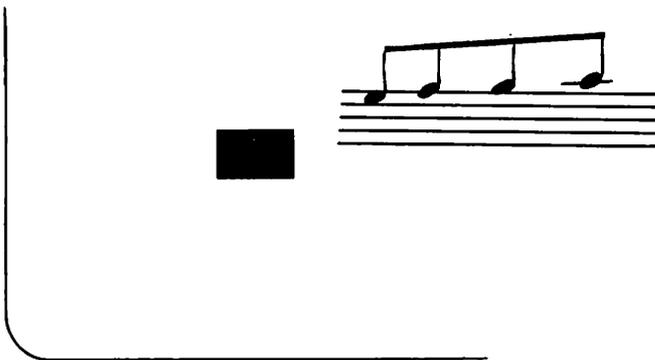
The last program just runs up the C scale and then starts over again. Let's add a little randomness to the program and allow it to select its own notes and colors. Also, we will change it so that the notes are played at varying durations. If we make the following changes, our new program will do all of these things.

```

100 T = INT(400*RND)+100 ← Set the duration from 100 to 500 milliseconds
130 ROW = INT(8*RND)+1 ← Pick a random row
220 GO TO 100 ← Loop back to get new note
230 (Just press the ENTER key) ← Erases line 230 from program

```

RUN this new program. What do you hear? The program should be playing a "selection" from the eight notes that it has available in the B array. The screen should be changing colors near the center as each note is played. The monitor must look something like this:



This simple "color organ" can be expanded upon, especially the graphics portion of the program. What would it be like to "paint" the screen with colors in time to the notes being played? Also, we could put the colors on in either a vertical or horizontal direction, and vary the length of the color bars being displayed. Want to give this program a try? OK! Let's go!

The READ-DATA section of the program remains the same (lines 10 – 80). We now add the lines:

```

172 H = INT(RND*24)+1
174 A = INT(RND*32)+1
176 N = INT(RND*20)+1
178 IF INT(RND*2) = 0 THEN 190
180 CALL HCHAR(H,A,42,N)
185 GO TO 150
190 CALL VCHAR(H,A,42,N)

```

← Pick a random horizontal and vertical location on the screen  
 ← Set number of characters to be printed (1 to 20)  
 ← Random selection of horizontal or vertical display  
 1 = HCHAR 0 = VCHAR

The complete program now looks like this:

```

10 FOR ROW = 1 TO 8
20 REM**LOAD FREQUENCY COLUMN 1**
30 FOR COL = 1 TO 2
40 READ B(ROW, COL)
50 REM**LOAD COLOR COLUMN 2**
60 NEXT COL
70 NEXT ROW
80 DATA 262,5,294,7,330,11,349,14
90 DATA 392,6,440,8,494,10,523,12
95 REM**SET PARAMETERS**
100 T = INT(400*RND)+100
110 V = 2
120 REM**GET FREQ AND COLOR**
130 ROW = INT(8*RND)+1
140 TONE = B(ROW,1)
150 C = B(ROW,2)
160 REM**DISPLAY BAR**
170 CALL COLOR(2,C,C)
172 H = INT(RND*24)+1
174 A = INT(RND*32)+1
176 N = INT(RND*20)+1
178 IF INT(RND*2) = 0 THEN 190
180 CALL HCHAR(H,A,42,N)
185 GO TO 150
190 CALL VCHAR(H,A,42,N)
200 REM**SOUND ON**
210 CALL SOUND(T,TONE,V)
220 GO TO 70

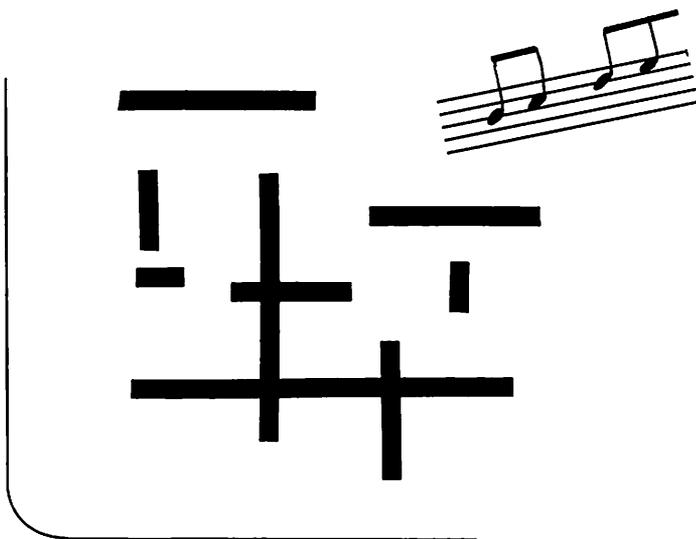
```

← Section to fill B array  
 ← Section that "paints" colors on screen and plays notes

Check to see if the program in your TI computer is the same as the one listed above. If it is, RUN it. What appears on the screen?

The screen will begin to look like this

Different colors everywhere!



It's time for you to experiment. Why don't you try changing the colors that are displayed? Or better yet, why not set up two arrays with two different musical scales? You could randomly pick a note from either scale and have the CALL SOUND routine play both notes at the same time.

The scale we have been using is a C-natural scale. It has eight tones. There are many other scales. Pythagoras, an ancient Greek, discovered a lot about music many years ago. He devised a 7-tone scale. Later, around the time of J.S. Bach, it became possible to construct and use twelve-tone scales. We can make our program work with twelve-tone scales. We just have to enter the following changes:

```

5 DIM B(12,2) ← We put a DIMension statement in the program. This is good
10 FOR ROW = 1 TO 12 practice most of the time since the DIM statement tells you
80 DATA 440,8,466,9,494,10,523,12,554,13,587,5 ← New frequencies and colors
90 DATA 622,7,659,11,698,14,740,6,784,2,831,15 ←
130 ROW = INT(12*RND)+1 ← Random selection of row in array

```

Let's take a breather from all this color, music and graphics and look at some other uses of two-dimensional arrays.

Arrays are useful to hold information that is related. For example, if we had a list of names of people and their telephone numbers, a single array could be used to hold this information. Having the data in a single array makes it easy to retrieve and display the data in tables.

We write a program that uses the array N\$ to hold the names and telephone numbers of five people that we know. The program loads the data into the N\$ array with a READ-DATA statement, and prints the information from the array in a table.

```

10 DIM N$(5,2)
20 REM**FILL THE ARRAY**
30 FOR ROW = 1 TO 5
40 FOR COL = 1 TO 2
50 READ N$(ROW,COL)
60 NEXT COL
70 NEXT ROW

75 DATA "HARRY","555-0707"
80 DATA "BILL","555-2223"
85 DATA "KATHY","555-3735"
90 DATA "MARY R.,""555-2134"
95 DATA "MARY Z.,""555-1796"

100 REM**PRINT THE TABLE**
110 CALL CLEAR
120 PRINT " NAMES ", " PHONES "
130 PRINT " "
140 PRINT

150 FOR ROW = 1 TO 5
160 PRINT N$(ROW,1), N$(ROW,2)
170 NEXT ROW
180 END

```

We will use a DIMension statement. Your TI Home Computer does not require one if the array dimensions are less than ten (10) in each position

Set up nested loops over rows and columns

Read the data. COL=1 is for names; COL=2 is for phone numbers

End the loops

Names and numbers

Print header on table

Display names and numbers

When this program is RUN, the screen will look like this:

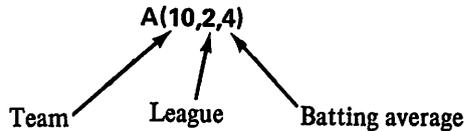
| ... NAMES ... | ... PHONES ... |
|---------------|----------------|
| HARRY         | 555-0707       |
| BILL          | 555-2223       |
| KATHY         | 555-3735       |
| MARY R.       | 555-2134       |
| MARY Z.       | 555-1796       |

Do you see how this program made use of two-dimensional string arrays? Can you think of other applications? There are as many as there are people who own TI Home Computers. Experiment with some variations of the last program for those applications you have an interest in seeing.

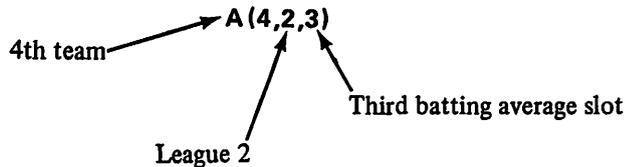
When you are ready, move on to the next part of this chapter, where we will be going into another *dimension* – three-dimensional arrays.



With the breakdown by batting averages, we get four two-dimensional arrays like the  $A(10,2)$  array. However, we can represent all of this information in a compact way. We can refer to one three-dimensional array by adding a third dimension to  $A$ . We would get:



The first dimension in the  $A$ -array points us to the team the player is on; the second dimension to the league; the third to the player's batting average. So if we had a player who played on team 4 of league 2, and had a batting average of 275, we would increase the contents of the array by one, for this player.



But wait! There might be a problem. What if a player has an average of 250? What location of the  $A$  array would he be counted into? Unless we relabel our array limits for batting averages, he would go into both batting average slots 2 and 3 of  $A$ . To correct this potential problem, we define the batting average slots to be:

| Third Dimension of A-array | Range of Batting Averages |
|----------------------------|---------------------------|
| 1                          | 199 and below             |
| 2                          | 200 to 249                |
| 3                          | 250 to 299                |
| 4                          | 300 and above             |

We now have an array defined that will hold the information that the boy wishes to store. The array has  $10 \times 2 \times 4$ , or 80 locations in which to place data items. Our first program dealing with this array shows how to fill it using the TI Home Computer to prompt for INPUTs.

```

10 REM**EXAMPLE 3-D ARRAY**
20 REM**BASEBALL CARD PROGRAM**
30 REM**DATA INPUT ROUTINE**
40 DIM A(10,2,4) ← Dimension A and clear screen
50 CALL CLEAR
60 PRINT "INPUT TEAM NO., LEAGUE NO. AND BATTING AVERAGE"
70 INPUT "T,L,BA--?": T,L,BA ← Input pointers T and L, and batting average (BA)
80 IF T = 0 THEN 200 ← If input is complete, enter 0,0,0
90 REM**THE NEXT FEW LINES COMPUTE THE SLOT NUMBER(B)**
100 REM**BASED ON THE BATTING AVERAGE(BA)**
110 B = 1
120 IF BA 199 THEN 130 ELSE 180
130 B = B+1 ← B is increased
140 IF BA 249 THEN 150 ELSE 180
150 B = B+1 ← B is increased
160 IF BA 249 THEN 170 ELSE 180
170 B = B+1 ← B is increased
180 A(T,L,B) = A(T,L,B)+1 ← Increment total players by one
190 GOTO 70 ← Return for next INPUT
200 REM**INPUT NOW COMPLETE**

```

The preceding program will accept inputs from the keyboard: information on the team, league, and player's batting average. It stores the data in the A-array. An entry of 0,0,0 in line 70 terminates the input. The program then branches to the next part of the program. What shall we build next?

Let's suppose our "user" wishes to print three kinds of statistics from the entered data: (1) the total number of players in each leagues, (2) the total number of players in each batting average slot, and (3) a summary table, by league, of teams and batting average breakdowns. Sounds like a lot, doesn't it? Well, let's just do it a piece at a time and see what happens.

First, let's add a section, beginning with lines 200, that allows us to find out which table the "user" wants.

```

200 REM**INPUT COMPLETE**
210 CALL CLEAR
220 PRINT "DO YOU WANT:"
230 PRINT "TOTALS BY LEAGUE(1)"
240 PRINT "TOTALS BY BATTING AVERAGE SLOTS(2)"
250 PRINT "SUMMARY TABLE BY LEAGUE(3)"
260 PRINT "EXIT THE PROGRAM(4)" ;
270 INPUT N
280 CALL CLEAR
290 ON N GOTO 300,500,700,1000

```

The fourth option, to exit the program, is easily handled by the following line:

```
1000 END
```

*Note:* This program and the associated data file are quite large. They are presented here as an example of how to use 3-D arrays. In practice, for large data files and large programs, you will need to start considering what is going to be the best way for you to create and save both data and programs. You would *not* want to reenter this program each time you wished to use it. You most likely would want to use a mass storage medium (tape, disk, etc.).

To get the totals by individual league, we enter the next set of program statements:

```

296 REM**ROUTINE TO CALCULATE LEAGUE TOTALS**
300 TOT = 0
310 FOR L = 1 TO 2
320 LTOT = 0
330 FOR B = 1 TO 4
340 FOR T = 1 TO 10
350 LTOT = LTOT + A(T,L,B)
360 NEXT T
370 NEXT B
380 PRINT "NUMBER OF PLAYERS IN LEAGUE ";L;" = "; LTOT
390 TOT = TOT + LTOT
400 NEXT L
410 PRINT "TOTAL PLAYERS IN BOTH LEAGUES = ";TOT
420 GOSUB 990
430 GOTO 210

```

Let's look at what this routine does:

- Line 300: Sets TOT to zero. TOT will contain the total number of players in both leagues at the end of execution of this routine.
- Line 310: Begin loop over the number of leagues.
- Line 320: Set LTOT to zero. LTOT will contain the number of players (total) for a particular league.
- Line 330: Begin loop over the number of batting average slots.
- Line 340: Begin loop over the number of teams.
- Line 350: Sum up players in this league.
- Line 360: End the team loop.
- Line 370: End the batting average slot loop.
- Line 380: Display the number of players in league L.
- Line 390: Add the number of players in league to total players.
- Line 400: End the league loop.
- Line 410: Display the total number of players.
- Line 420: Delay routine call.
- Line 430: Return to the "menu" of table options.

For the totals by batting average slots, we enter:

```

495 REM**ROUTINE TO CALCULATE TOTALS BY BATTING AVERAGES**
500 TOT = 0
510 FOR L = 1 TO 2
520 PRINT "LEAGUE ";L
530 PRINT "
-----"
540 FOR B = 1 TO 4
550 BTOT = 0
560 FOR T = 1 TO 10
570 BTOT = BTOT + A(T,L,B)
580 NEXT T
590 PRINT "SLOT "; B; " # PLAYERS = "; BTOT
600 TOT = TOT + BTOT
610 NEXT B
620 PRINT
630 NEXT L
640 PRINT "TOTAL PLAYERS IN BOTH LEAGUES = "; TOT
650 GOSUB 990
660 GO TO 210

```

This routine does the following:

- Line 500: Set TOT to zero. TOT will contain total number of players.
- Line 510: Begin league loop.
- Line 520: Begin displaying header message for league.
- Line 530: Decorations.
- Line 540: Begin batting average loop.
- Line 550: Set BTOT to zero. BTOT is variable that will contain the total number of players in a particular slot.
- Line 560: Begin team loop.
- Line 570: Sum players by batting average, across all teams.
- Line 580: End team loop.
- Line 590: Display batting average slot number, and players.
- Line 600: Add partial sum of players to total sum.
- Line 610: End the batting average loop.
- Line 620: Display a blank line.
- Line 630: End the league loop.
- Line 640: Display the total number of players in the league.
- Line 650: Delay routine call.
- Line 660: Return to "menu" of table options.

The last routine is a bit longer. It provides the summary table, by league, of the number of players that are on each team and in each batting average category. Here is the routine:

```

695 REM**SUMMARY TABLE ROUTINE**
700 INPUT "INPUT LEAGUE NUMBER(1,2)?":L
710 PRINT "SUMMARY TABLE FOR"
720 PRINT " LEAGUE ";L
730 PRINT" BATTING AVERAGE"
740 PRINT " SLOT SLOT SLOT SLOT"
750 PRINT " 1 2 3 4"
760 PRINT " "
770 PRINT

```

} Header for table

```

780 FOR B = 1 TO 4
790 STOT(B) = 0 ← Clear the STOT array
800 NEXT B
810 FOR T = 1 TO 10
820 PRINT "TEAM"; T;
830 TB = 10
840 FOR B = 1 TO 4
850 STOT(B) = STOT(B) + A(T,L,B)
860 IF B = 4 THEN 890
870 PRINT TAB(TB); A(T,L,B);
880 GO TO 900
890 PRINT TAB(TB); A(T,L,B)
900 TB = TB + 5
910 NEXT B
920 NEXT T
930 PRINT
940 PRINT "TOTALS";TAB(10);STOT(1);TAB(15);STOT(2);TAB(20);
STOT(3);TAB(25);STOT(4)
950 GOSUB 990
960 GO TO 210
990 FOR T = 1 TO 3000
992 NEXT T ← Delay routine
994 RETURN

```

Here is what the last routine does when it is executed:

- Line 700:                Requests an input of the league number.
- Lines 710 to 770:      Display a header message for the table.
- Lines 780 to 800:      Clear the STOT array. STOT is used to store the totals that are printed at the bottom of the table.
- Lines 810 to 920:      Computes the totals, and displays the body of the table. The variable TB is used to position the columns of the table. It is the parameter for the TAB function calls.
- Lines 930 to 950:      Print a line of totals for the columns of the table, and return the user to the "menu" of table options, after a delay.
- Lines 990 to 994:      The delay routine.

If you have entered this entire program, *take a break!* Your fingers must be tired. When you have rested, RUN the program. If you do, this is what happens on the screen.

First, the screen clears and the messages in lines 60 and 70 appear on the monitor.

```
INPUT TEAM#, LEAGUE# AND BATTING AVERAGE
T,L,BA- -?□
```

You then enter triplets of numbers representing the team number, league number, and batting average for each player:

```
INPUT TEAM#, LEAGUE# AND BATTING AVERAGE
T,L,BA- -73,2,240
T,L,BA- -74,1,301
T,L,BA- -? ↑ ↑ ↑
 Team # League # Batting Average
```

You would continue to enter data until all the information you wanted to tabulate was in the computer. Then you type 0,0,0 to display the table.

When 0,0,0 is entered, the screen clears and the “menu” of table options appear:

```

DO YOU WANT:
TOTALS BY LEAGUE(1)
TOTALS BY BATTING AVERAGE SLOTS(2)
SUMMARY TABLE BY LEAGUE(3)
EXIT THE PROGRAM(4)?

```

If you enter a one (1), the screen again clears. The totals, by league, are calculated, and the screen shows:

```

NUMBER OF PLAYERS IN LEAGUE 1 = 155
NUMBER OF PLAYERS IN LEAGUE 2 = 160
TOTAL PLAYERS IN BOTH LEAGUES = 315

```

The computer delays for a moment to let you see the table. After the pause, the “menu” appears once again on the screen.

If you enter a two (2), the following information is displayed:

```

LEAGUE 1

SLOT 1 #PLAYERS = 30
SLOT 2 #PLAYERS = 50
SLOT 3 #PLAYERS = 60
SLOT 4 #PLAYERS = 15

LEAGUE 2

SLOT 1 #PLAYERS = 25
SLOT 2 #PLAYERS = 55
SLOT 3 #PLAYERS = 65
SLOT 4 #PLAYERS = 15

```

↖  
Total players in both leagues = 315

Here again, the screen holds the information for you to look at, then clears and redisplay the “menu.” If you now enter a three (3), the last table is displayed:

First you are asked for the league number

INPUT LEAGUE NUMBER(1,2)?

When you enter the league number, the header message for the table and the table begin to appear:

```

INPUT LEAGUE NUMBER(1,2)?1
SUMMARY TABLE FOR
LEAGUE 1
 BATTING AVERAGE
 SLOT SLOT SLOT SLOT
 1 2 3 4

TEAM 1 3 5 7 1
TEAM 2 4 7 5 2
TEAM 3 2 5 5 1
TEAM 4 1 3 8 1
TEAM 5 5 2 6 3
TEAM 6 3 9 6 0
TEAM 7 3 1 3 2
TEAM 8 4 8 9 1
TEAM 9 2 4 7 2
TEAM 10 3 6 4 2

TOTALS 30 50 60 15

```

Once more, the computer pauses while you scan the table, then branches back and displays the “menu” again. If you enter a four (4) when the “menu” is on the screen, the program stops executing.

This concludes the discussion on multidimensional arrays. You will have many uses for these convenient variables as you develop more of your own applications.

**Chapter Ten Exercises**

The first six exercises relate to the two-dimensional array, B, shown below.

B(ROW,COL)

Columns

|   | 1  | 2  | 3  |
|---|----|----|----|
| 1 | 7  | 2  | 8  |
| 2 | 1  | 5  | 4  |
| 3 | 20 | -6 | 1  |
| 4 | 11 | 9  | 12 |
| 5 | 19 | 10 | 6  |

Rows

B is a 5 by 3 position array (table).

(1) What are the values in the following B-array locations?

- (a)  $B(3,1) =$  \_\_\_\_\_
- (b)  $B(5,3) =$  \_\_\_\_\_
- (c)  $B(3,2) =$  \_\_\_\_\_
- (d)  $B(1,3) =$  \_\_\_\_\_

(2) What are the location names that contain these data values?

- (a) 4 \_\_\_\_\_
- (b) 11 \_\_\_\_\_
- (c) 5 \_\_\_\_\_
- (d) 10 \_\_\_\_\_
- (e) 1 \_\_\_\_\_ and \_\_\_\_\_

(3) Write the program statement below that would dimension the B-array.

10 \_\_\_\_\_

(4) For matrix location B(5,1), what are the results of the next set of statements:

- (a)  $B(5,1) = B(2,1) + B(1,2)$  \_\_\_\_\_
- (b)  $B(5,1) = B(3,2) + B(2,3)$  \_\_\_\_\_
- (c)  $B(5,1) = B(2,2) * B(2,3)$  \_\_\_\_\_
- (d)  $B(5,1) = B(4,3)/B(2,3)$  \_\_\_\_\_

- (5) Describe what happens with this program, assuming the B-array contains the data we started with in Problem 1:

```

10 FOR ROW = 1 TO 5
20 SUM = 0
30 FOR COL = 1 TO 4
40 SUM = SUM + B(ROW, COL)
50 NEXT COL
60 PRINT "SUM FOR ROW"; ROW; " = "; SUM
70 NEXT ROW

```

---



---



---

- (6) Write a DATA statement that could be used to read in the B-array.

```
100 DATA _____
```

- (7) Suppose we are writing a program that needs an array with the properties:

One dimension represents *seven* classifications by age.  
 A second dimension represents *ten* weight categories.  
 A third dimension represents any one of *twenty* geographic locations.

If we call the array C, what does the dimension statement look like?

```
10 _____
```

- (8) For the array in Problem 7, write a set of loops that would total all the elements of the C-array, and place the total in the variable TOT.

```

10 DIM _____
20 TOT = _____
30 FOR AGE = _____
40 FOR WEIGHT = _____
50 FOR LOC = _____
60 TOT = TOT + C(_____)
70 NEXT _____
80 NEXT _____
90 NEXT _____

```

- (9) Change the program in Problem 8 so that it totals all the elements in the C-array within any particular age classification.

(a) Change Line 30:    **30 INPUT** \_\_\_\_\_

(b) Delete line \_\_\_\_\_

## Chapter Ten Solutions

- (1) (a) .20    (b) 6    (c) -6    (d) 8
- (2) (b) B(2,3)    (b) B(4,1)    (c) B(2,2)  
 (d) B(5,2)    (e) B(2,1) and B(3,3)
- (3) 10 DIM B(5,3)
- (4) (a) B(5,1) = 3    (b) B(5,1) = -2    (c) B(5,1) = 20    (d) B(5,1) = 3
- (5) The program totals each row of the B-array, putting the total in SUM. When a row is totaled, across all columns, a message and SUM is printed on the screen. The screen should look like this:

```
SUM FOR ROW 1 = 17
SUM FOR ROW 2 = 10
SUM FOR ROW 3 = 15
SUM FOR ROW 4 = 32
SUM FOR ROW 5 = 35
```

- (6) If read by rows:    100 DATA 7,2,8,1,5,4,20,-6,1,11,9,12,19,10,6  
 If read by columns:    100 DATA 7,1,20,11,19,2,5,-6,9,10,8,4,1,12,6
- (7) 10 DIM C(7,10,20)
- (8) 10 DIM C(7,10,20)  
 20 TOT = 0  
 30 FOR AGE = 1 TO 7  
 40 FOR WEIGHT = 1 TO 10  
 50 FOR LOC = 1 TO 20  
 60 TOT = TOT + C(AGE,WEIGHT,LOC)  
 70 NEXT LOC  
 80 NEXT WEIGHT  
 90 NEXT AGE
- (9) (a) 30 INPUT "AGE CATEGORY(1-7)":AGE    Or something similar.  
 (b) Delete line 90.

# Chapter Eleven

## Color, Graphics, Sound, and Animation

In preceding chapters, you were introduced to some of the musical and graphic capabilities of your TI Home Computer. What we want to do now is to expand your knowledge in these areas.

It is possible, on your computer, to create your own graphics characters. As you develop programs and simulations, you may discover that you would like to add some color, sound, or graphical accents that go beyond the CALL COLOR, CALL SOUND, and CALL VCHAR/HCAR applications discussed so far.

This chapter gives you hints and examples on how to approach an expanded use of color and graphics. By using this material, you will begin to discover techniques for making programs visually exciting. You can have images “moving” about the screen. They can be changing colors. Appropriate sounds can accompany the movements. Sound interesting? You bet it is! These capabilities of your TI Home Computer add many new dimensions to what is possible with these small machines.

We will introduce three new routines: CALL SCREEN, CALL CHAR, and CALL KEY, each of which expands your ability to create colorful graphics with your machine. In brief, here is what each of the routines does:

- CALL SCREEN – Allows you to change the color of the entire screen.
- CALL CHAR – Can be used to create new graphics characters.
- CALL KEY – Lets you input data without pressing ENTER.

### The CALL SCREEN Statement

When a program is running, the background screen color is light green. Up to this point, there has been no easy way to change the color of the screen completely while in the Run Mode. Now we can do so. We use the following statement:

**CALL SCREEN (C)**

Color code for the screen  
color we want goes here

Clear the machine's memory by typing NEW, and enter the small program given below:

```

10 REM**CHANGING THE SCREEN COLOR **
20 CALL CLEAR
30 INPUT "SCREEN COLOR CODE?": C
40 IF C < 1 THEN 20 ← Check for
50 IF C > 16 THEN 20 ← valid code
60 CALL SCREEN (C) ← Change screen color
70 INPUT "PRESS ENTER TO CONTINUE": A$ ← from light green
80 GO TO 20 ← to color you input

```

RUN the program. The request for a color code will appear, and the screen. If you enter any color code (except 4, which is light green), the screen will change to that color. The program pauses at line 70 so you can see the color. Pressing the ENTER key causes the program to loop back and request a new color code.

What happens when you enter a color code of one (transparent) or color code two (black)? Try it and see!

### Introducing CALL CHAR

The CALL CHAR statement gives you the capability of creating your own characters on the screen. One way to do this is to *redefine* one of the standard characters to be the new character that you want. Thus, when the program attempts to display the standard character, your new character, appears on the screen.

The redefinition of a standard character is handled by the CALL CHAR routine. Suppose we wish to redefine the asterisk (character code 42). We do so by putting a statement in our program that looks like this:

```

CALL CHAR (42, A$)

```

Code for character we are redefining →

← A string variable that will contain the "code" for our new character.

For now, we are going to use just the CALL CHAR routine without going into a lot of detail on how it works. Later in the chapter we will provide an in-depth discussion.

The string variable parameter, A\$ in the CALL CHAR statement contains a set of letters and numbers that get translated into a new graphics character. For now, enter the next program and RUN it.

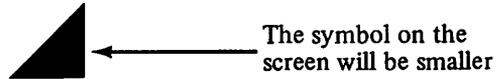
```

10 REM** DISPLAY A NEW CHARACTER **
20 A$ = "0103070F1F3F7FFF" ← The "code" for the new character
30 CALL CHAR (42, A$) ← Redefine the
40 CALL CLEAR ← '*'
50 CALL VCHAR (12, 16, 42) ← Display new character near
60 GO TO 60 ← center of screen
Wait here!

```

What do you see? A small triangular figure in the center of the monitor? Yes! We have created a new character!

The character looks something like this:



Now, press the **SHIFT C** to stop the program. What's on the screen after you do this? That's right, the asterisk (\*) appears in the place of the new symbol.

The **CALL CHAR** routine redefines the standard character, the asterisk, to be the new symbol while the program is running. When the program stops, the redefinition is canceled, and the standard character appears wherever the new symbol was displayed on the monitor.

We know that you have questions concerning the long, strangelooking "code" that is put into **A\$**. Hold those questions for a while, and let's use the **CALL CHAR** routine in a couple of programs. We'll cover the questions later in this chapter.

## COLOR with CHAR

We've experimented before with programs that put color on the screen. We now want to combine our new capability of creating a graphics character with the use of color displays. The next program creates larger blocks of triangular symbols (redefined asterisks), and let's you specify what color they are to be.

```

10 REM**COMBINING COLOR AND CHAR**
20 CALL CLEAR
30 A$ = "0103070F1F3F7FFF" ← Redefine asterisk
40 CHAL CHAR (42, A$) ←
50 INPUT "COLOR CODES?": F, B
60 CALL CLEAR
70 CALL COLOR (2, F, B) ← Set color
80 REM**SHOW COLORS**
90 FOR Y = 1 TO 4
100 CALL VCHAR (19, Y+2, 42, 4)
110 CALL VCHAR (19, Y+2, 42, 4)
120 CALL VCHAR (2, Y+24, 42, 4)
130 CALL VCHAR (19, Y+24, 42, 4)
140 CALL VCHAR (12, Y+13, 42, 4)
150 NEXT Y
160 INPUT "PRESS ENTER TO CONTINUE":B$ ← Wait here
170 GO TO 50

```

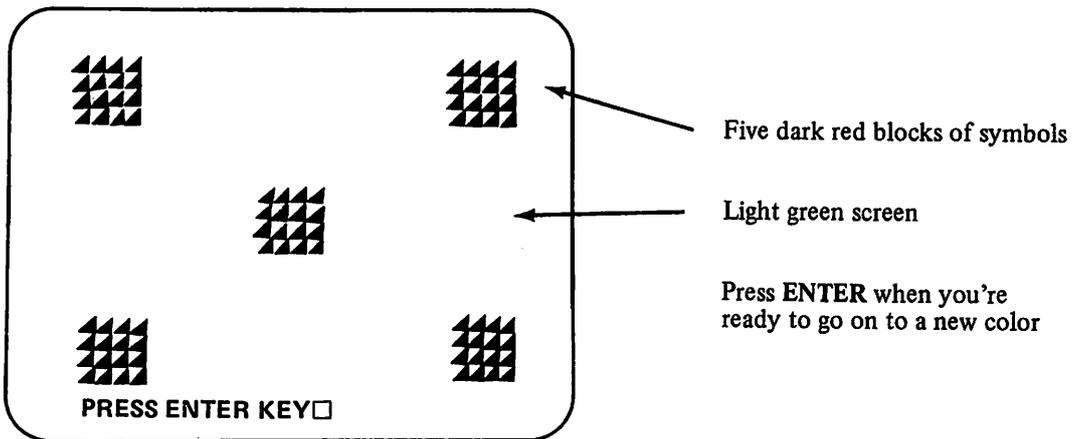
Each **CALL VCHAR** places a vertical strip of color on the screen

The program begins by redefining the asterisk to be the new graphics symbol. A request is then made for a color code. When you enter a code, the screen clears, the colored blocks of symbols are displayed, and the program waits for you to press the **ENTER** key. When you press **ENTER**, the program goes back and asks for a new color code.

Clear the screen and RUN the program. First you'll see:

COLOR CODES? □ ← Flashing cursor

Try a 7 and 4 (dark red and light green) for the first color codes. Wow! The screen should look like this:



Examine one of the blocks of symbols closely. What do you see? There should be a pattern made up of triangular symbols and triangular "holes." Try various color combinations. Which colors give the sharpest, clearest design?

Here are two interesting experiments. Delete line 40 from the program (the line that redefines the standard character). Run the program with the same color for the foreground and background. What happens? Yes, solid blocks of color appear on the monitor. Can you picture how this program could be modified to produce checkerboards or colored playing areas for tic-tac-toe?

A second experiment could involve the addition of the CALL SCREEN routine to the program. This addition would allow you to vary the background of the entire screen as you changed the colors of the blocked symbols.

Try these and other variations on your own. Become a color "artist" with your new Home Computer.

## PRINTed Patterns

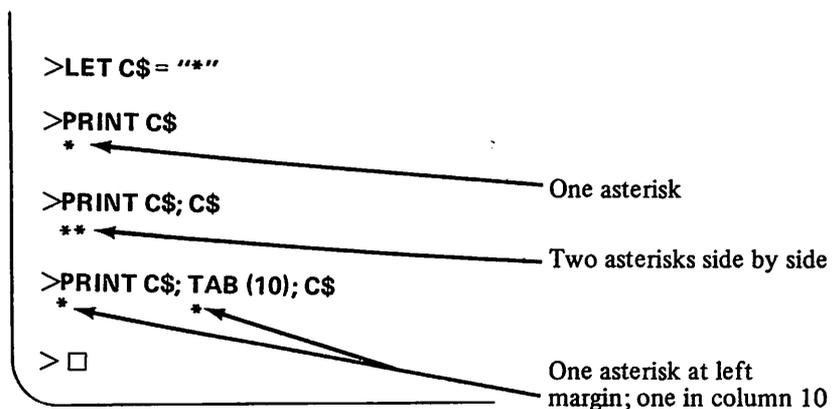
We will now show you a set of programs that can be used to create blocks of patterns on the screen. The programs use the string variable features of the TI Home Computer and the familiar PRINT statement. The first program demonstrates how to construct squares or rectangles of standard characters.

### Rectangles and Squares

The first program allows you to place a rectangle or square of standard characters on the screen. Instead of using CALL HCHAR or CALL VCHAR and identifying the character by its *character code* (numbers 32 through 95), we'll assign a character to a string variable from the keyboard.

Try these examples in the Immediate Mode:

```
LET C$ = "*"
PRINT C$
PRINT C$; C$
PRINT C$; TAB (10); C$
```



Try a few more Immediate Mode experiments on your own. For example, what would happen if you redefined C\$ as "\*\*\*\*" or as "(")? Try it and see what results!

This method is convenient if you want to print only a short line of characters. But what if you want to print a long line or vary the line length or character the program prints? INPUT statements and a FOR-NEXT loop will solve the problem. Type NEW; then enter this program:

```

10 REM**RECTANGLES AND SQUARES **
20 INPUT "CHARACTER?": C$ ← Accept character
40 INPUT "WIDTH?": W ← Accept number of
 characters to be printed
60 CALL CLEAR
80 FOR X = 1 TO W
100 PRINT C$; ← Semicolon
120 NEXT X
140 END

```

When you run the program, you'll first be asked to input the character you want to use. Just type the character and press **ENTER**. Then you'll be asked for the "width" or the number of characters in the line you want to print. Type in the number and press **ENTER** to continue the program. Let's say that you entered \* as the character and 28 as the width. The screen will look something like this.

```

** DONE **
> □

```

(Note that the semicolon in line 100 causes the characters to be printed in an unbroken row.)

Run the program a few times, entering different characters and lengths. Now let's try adding some program lines that will allow us to make rectangles and squares of characters.

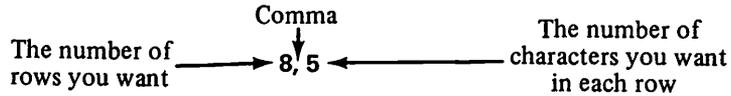
Enter these new lines:

```

40 INPUT "SIZE (WIDTH, HEIGHT)": W, H ← Replaces old line 40
70 FOR Y = 1 TO H
130 PRINT
135 NEXT Y
140 GO TO 40 ← Replaces old line 140

```

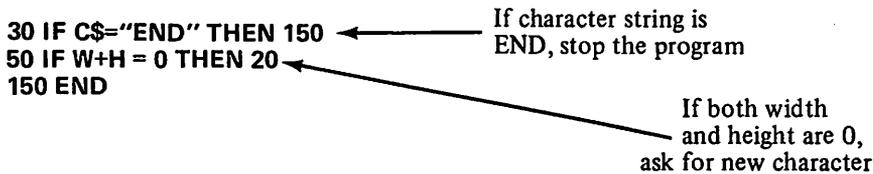
There are a couple of items that need to be explained about these lines. First, notice in line 40 that we are using one INPUT statement to assign values to *two* variables ! When you input the width and height, you'll need to use this form:



Second, lines 70 and 135 set up a loop on the variable Y. Your original "X loop" is now nested inside the "Y loop."

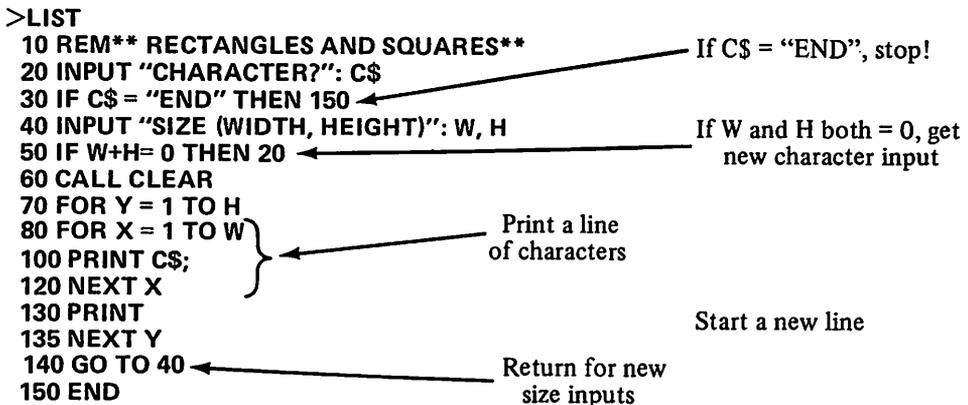
Finally, line 130 prints an "empty" line. This line is needed to clear away the semicolon (;) in line 100 so that a new row will begin the next time the program loops through the "Y loop." (As you've seen already, the semicolon causes the characters to be printed on the same line throughout the loop on X.)

Before we list the program to see the changes, let's add a few more lines. We can use IF-THEN statements to "build in" some tests:

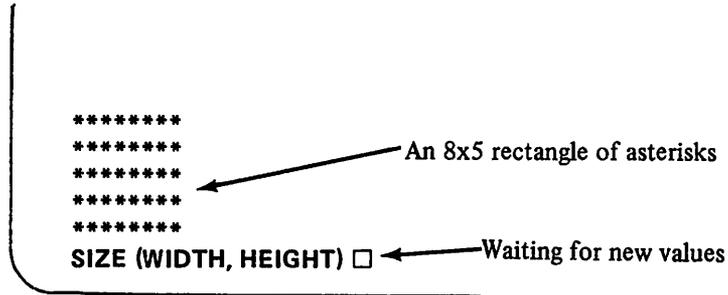


Here's what these tests provide. Line 30 gives you a handy way to stop the program by typing END, pressing the X key twice and then pressing ENTER when you're asked for a character input. If you want to experiment with a different character, all you have to do is to enter 0,0 as size inputs. The test in line 50 then sends you back to line 20 to input a new character.

Now clear the screen and list the program:



Clear the screen again and run the program. For this example, enter \* when the program asks CHARACTER? Then enter 8, 5 when you're asked for width and height:



Next, enter the same value for both width and height, such as 8,8 or 5,5. With these inputs the program will create a square rather than a rectangle.

**“Triangular” Rectangles and Squares**

Let’s take the last program and change it. Let’s add a CALL CHAR that redefines the asterisk (\*) to be our triangular symbol. Also, let’s have the program produce a random sprinkling of holes (blank spaces) in the displayed pattern. We can use RANDOMIZE and the RND function to make this last part happen.

Enter this altered program:

```

10 REM**RECTANGLES AND SQUARES WITH HOLES AND TRIANGLES**
20 A$ = "0103070F1F3F7FFF"
30 CALL CHAR (42, A$)
40 RANDOMIZE
50 INPUT "CHARACTER?": C$
60 IF C$ = "END" THEN 210
70 INPUT "SIZE (WIDTH, HEIGHT)": W, H
80 IF W+H = 0 THEN 50
90 CALL CLEAR
100 REM**PRINT CHARACTER OR BLANK**
110 FOR Y = 1 TO H
120 FOR X = 1 TO W
130 IF INT(2*RND) = 0 THEN 160
140 PRINT " ";
150 GO TO 170
160 PRINT C$;
170 NEXT X
180 PRINT
190 NEXT Y
200 GO TO 70
210 END

```

Annotations for the code block:

- Arrow from line 130: INT(2\*RND) produces a 0 or a 1
- Arrow from line 130: If true, go to line 160 and PRINT the character. If false PRINT a blank space (line 140)
- Arrow from line 140: One space enclosed in quotation marks and followed by a semicolon
- Arrow from line 180: Skip line 160

The IF test in line 130 contains INT(2\*RND). This last expression randomly produces either a 0 or a 1. When it is zero, then the character is displayed. When it is a one, a blank space is placed on the screen. Approximately half the time the program prints a character; half the time it prints a space. RUN the program and observe the patterns that emerge. What happens when you enter an asterisk (\*) at the request for a character?

## Animation and Sound

In chapter 5, you discovered the way to create the illusion of movement on the screen. You used small programs that flashed characters on and off on the monitor so that they appeared to “dance” or move. The next program reworks one you did in Chapter 5 to include both the CALL CHAR and CALL SOUND features of your machine.

We use the CALL CHAR routine again to redefine the asterisk (\*) to be the triangular symbol. We place the character near the center of the screen. The FOR-NEXT loop causes a delay. At the end of this delay, a sound is emitted from the program, the screen clears, and a second delay is executed. The combination of the two delays creates the “flashing” of the character. Finally, a second sound is produced, and the program loops back to put the character on the screen again. Here is the program. Enter it and RUN it.

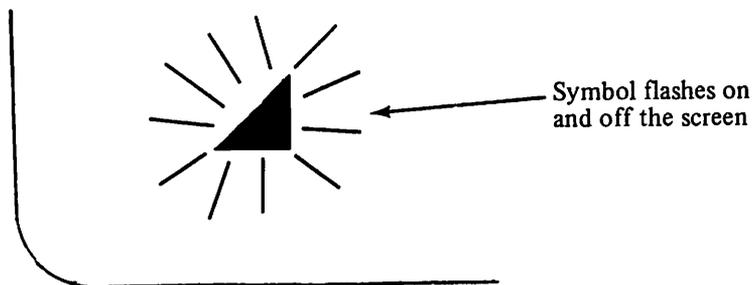
```

10 REM**SIMPLE ANIMATION WITH SOUND**
20 A$ = "0103070F1F3F7FFF"
30 CALL CHAR (42, A$)
40 CALL VCHAR (12, 16, 42) ← Numeric code for *
50 FOR DELAY = 1 TO 200 } ← Pause after printing
60 NEXT DELAY
70 REM**PLAY THE SOUND**
80 CALL SOUND (100, 110, 1)
90 CALL CLEAR
100 FOR DELAY = 1 TO 100 } ← Delay after cleaning
110 NEXT DELAY
120 CALL SOUND (100, 220, 1)
130 GO TO 40 ← Repeat

```

Play tones →

The triangular symbol should be flashing on the screen, and two distinct sounds should be occurring as the symbol appears and disappears.



This program is an excellent one to try some experiments on. Change the calls on the sound routine, and use “noise” instead of tones. Vary the duration of the noise being generated. Can you make the program emit “clicks,” “knocks,” and other interesting sounds? How about the sound a ball makes when it hits the paddle in a game? Does it make a different sound when the ball hits a wall?

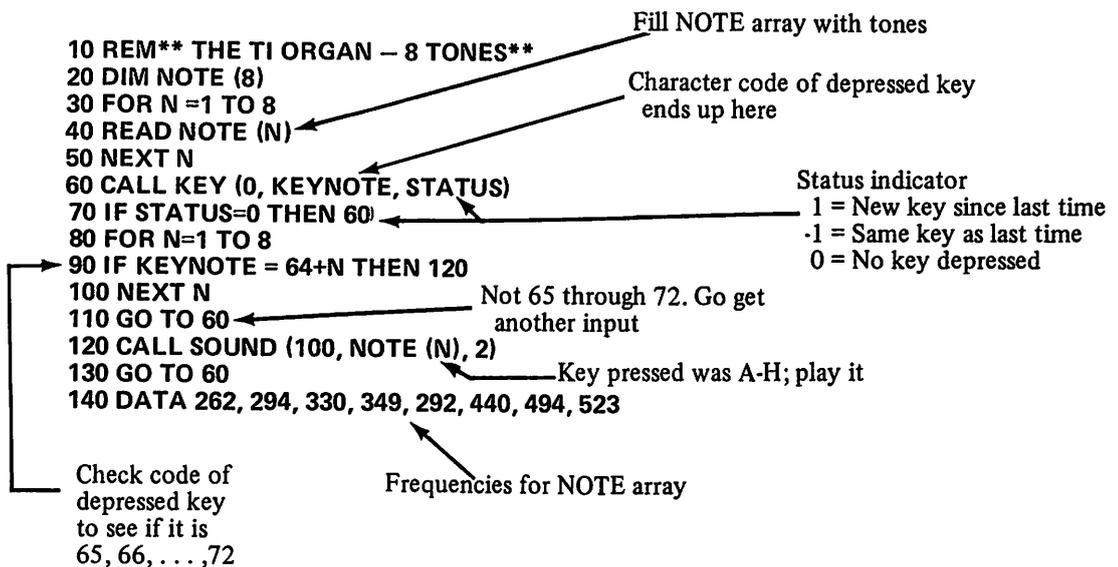
Can you see (and hear) that this small program represents a way that you can introduce sound into “animated” programs. All the interesting sounds you have heard in other computer games and simulations can be reproduced on your TI Home Computer. Give it a try!

## Musical Interlude

Break time from all this graphics stuff!! How about a little music to soothe the nerves and rest the spirit? Great!

We are going to revisit our tone-playing program, but with a new twist. With all the programming tricks you know by now, we can make the program really compact. We can also introduce you to one new feature of the TI Home Computer – the CALL KEY routine.

In previous programs, you had to enter the letter of the note to be played, and press the ENTER key. This double-keying action interrupted your musical “creations.” CALL KEY is used to transfer one character, from the keyboard, *directly* into the computer. You don’t have to press the ENTER key! Let’s enter the program, list it, and then explain how CALL KEY works.



Here is how CALL KEY works in this program. Each character on the keyboard has a numeric code. (You have used the code 42, the asterisk, several times earlier in this chapter.) When a key is depressed, the character code for that key is assigned to the second variable in the CALL KEY routine. In this example, the character code is assigned to the variable KEYNOTE.

The last variable in the routine is a status indicator. The indicator lets the program “know” what is occurring on the keyboard. If you press the same key twice, the status indicator, STATUS, is set to -1. If you press a different key on the second time through the routine, STATUS is set to 1. If no key is pressed, STATUS is set to 0.

When you RUN the program, nothing appears on the screen as you hit the keys. The program just plays the note you request. Try it! Make a little music. You will have to press the **SHIFT C** key to stop the program.

The **CALL KEY** routine allows you to create “your own kind of music.” The routine can also be used in many games and simulations where single character input values are requested. The effect of using the **KEY** routine is to speed up data input by eliminating the need to press **ENTER** each time.

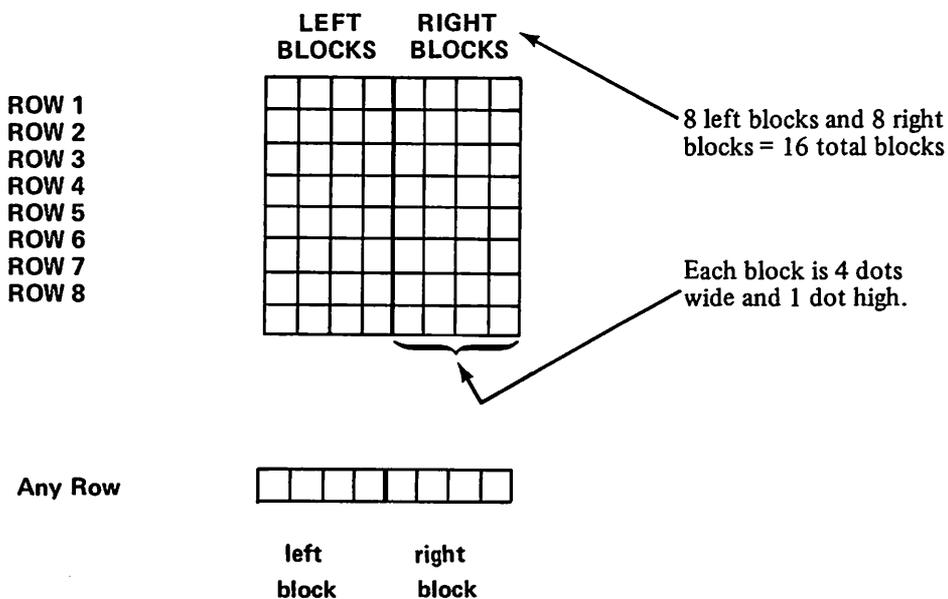
You may want to augment this program in many ways. How would you alter the program to accept sharp as well as natural notes? What would you need to do to allow yourself to key in a range of durations for each note? This one program is the basic idea from which you can begin to exploit fully the musical capabilities of your machine.

Perhaps, someday, there will be an organ keyboard that will plug directly into your Home Computer. We bet it will happen soon!

### The **CALL CHAR** Statement

You have used the **CALL CHAR** routine to redefine a standard screen character to be a special graphics symbol. We now want to look at the **CALL CHAR** feature in some detail. We begin by examining how a character, any character, is represented on the screen.

Each printing position on the screen is made up of 64 tiny dots. The dots are arranged in eight rows of eight dots each. Each row is partitioned into two blocks of four dots each. The diagrams below show how an 8-by-8 grid of dots would look if it were greatly enlarged.



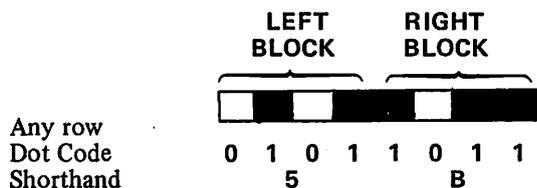
A character on the screen, either a standard character or one that you invent, is formed by dots within the 8-by-8 grid. By turning some dots "on" and leaving others "off" a character is created. Leaving all the dots "off" creates the space character (character code 32) for example. Turning all the dots "on" produces a solid spot on the screen.



All the standard characters are automatically set so that they turn on the appropriate dots to produce the images you have seen. To create a new character, we must tell the computer which dots to turn on or leave off within a particular block. The following table contains all the possible on/off conditions for the dots within a given block and the shorthand notation for each condition.

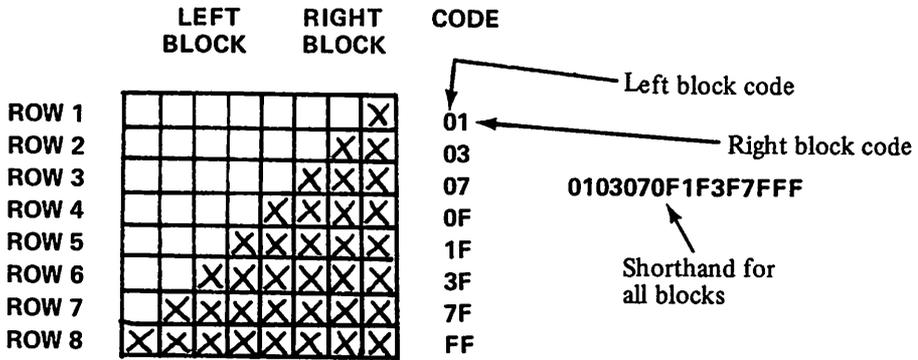
| BLOCKS | DOT CODE<br>(0 = off; 1 = on) | SHORT<br>HAND<br>CODE |
|--------|-------------------------------|-----------------------|
|        | 0000                          | 0                     |
|        | 0001                          | 1                     |
|        | 0010                          | 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                     |

Let's take a look at one row (two blocks) to see how the "shorthand code" works.



The shorthand code for the row, then, is 5B.

The shorthand codes for an entire grid can be determined block by block, just by converting the on/off conditions of each row. The following example provides a translation of an entire grid into the shorthand code.



The symbol in the grid and the shorthand code are those used in the early parts of this chapter. This code shows how the string was developed that produced the triangular symbol in those early programs.

Therefore, if we want to “define” a character shaped as the X’s on the grid indicate, we enter all the shorthand codes of the blocks as a single “string”:

**“0103070F1F3F7FFF”**

In the shorthand code, then, one number or letter represents a whole block (4 dots) on the grid. Two letters and/or numbers represent a whole row.

Based on the table, if all the dots in all the blocks were to be turned on, the shorthand code for this condition would be:

**“F F F F F F F F F F F F F F F F”** ← One F for each block

This code may seem long, since it represents all 16 blocks within the grid. But it is still shorter than trying to write down all 64 separate conditions dot by dot.

Once you’ve decided which dots you want on and off and worked out the code, you’re ready to use the CALL CHAR statement. It looks like this:

**CALL CHAR(33, “F F F F F F F F F F F F F F F F”)**

Code for character you are redefining

Comma

“String” that turns the dots on and off

Let's try a simple program that redefines a character code 33(!) as a character with all the dots turned on. The new character is then printed in the center of the screen, giving you a chance to see exactly how big one of the individual print areas really is. The program loops until you enter **SHIFT C** to cause it to stop. Enter these lines:

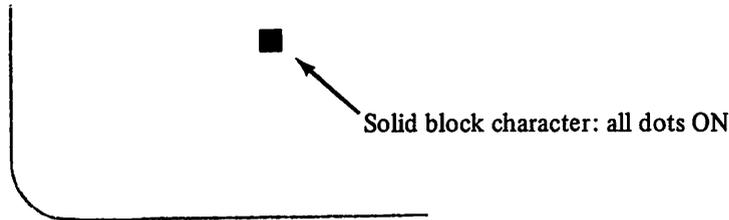
```

NEW
10 REM**CHAR TEST**
20 CALL CLEAR
30 CALL CHAR (33,"F F F F F F F F F F F F F F F F ")
40 CALL VCHAR (12, 16, 33)
50 GO TO 50

```

Redefining this character  
 String of shorthand codes  
 New character 33 in center of screen  
 What does this do?

Run the program and observe your newly defined character on the screen!



So that you can experiment with other shorthand codes, let's edit the program. Type these new lines:

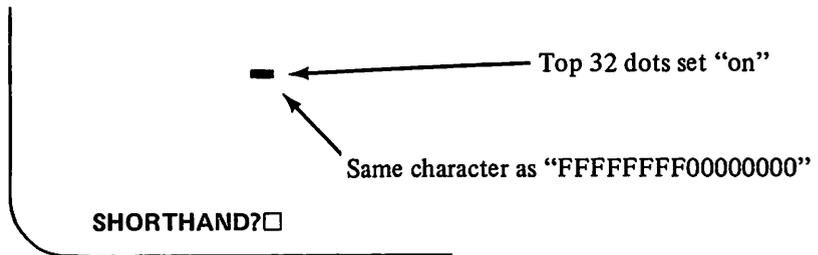
```

15 INPUT "SHORTHAND?": A$
30 CALL CHAR (33, A$)
50 GO TO 15

```

This time when you run the program, you'll be asked to input the shorthand code for the character you are redefining. Try the following examples.

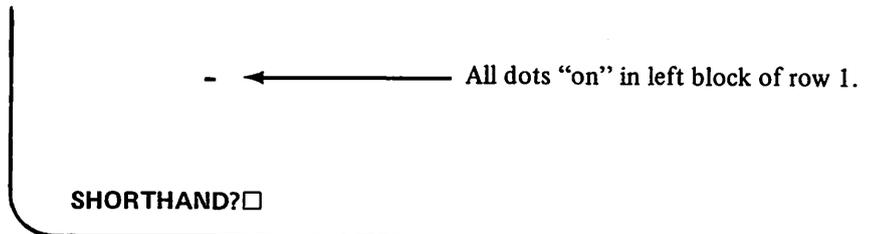
Enter: FFFFFFFF



When you stop the program by pressing **SHIFT C**, the character you created changes back into the character from the standard character set. In this case, character code 33 is restored to an exclamation point (!), and that symbol appears near the center of the screen.

Entering FFFFFFFF is the same as entering FFFFFFFF00000000. That is, the CHAR routine fills out the right side of the string variable with zeros when there are less than 16 characters in the string. Knowing this fact allows you easily to examine all the shorthand codes individually. Just enter 0, 1, and so on up to F at the INPUT request.

Enter: F



Try different combinations of the shorthand codes. See if you can generate any interesting characters. Now let's revise the program again to print more than just one of our redefined characters. Enter these lines:

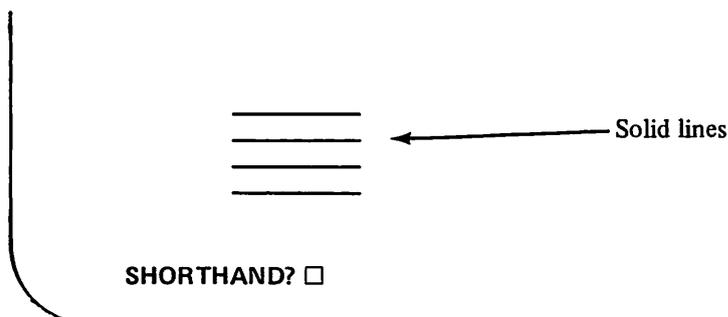
```
40 FOR I = 1 TO 4
50 CALL VCHAR (12, I+13, 33, 4)
60 NEXT I
70 GO TO 15
```

Now list the program to see the changes:

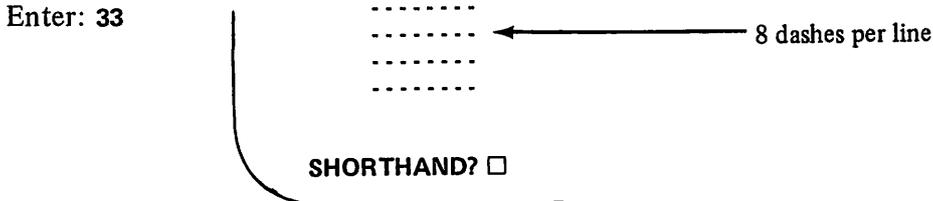
```
LIST
10 REM**CHAR TEST**
15 INPUT "SHORTHAND?": A$
20 CALL CLEAR
30 CALL CHAR (33, A$)
40 FOR I = 1 TO 4
50 CALL VCHAR (12, I+13, 33, 4)
60 NEXT I
70 GO TO 15
```

When you run this program and enter a shorthand code for a new character, that character is displayed 16 times in the center of the screen. The 16 characters appear in a square four characters wide by four characters high. Try the following:

Enter: FF



A single print of the character with the shorthand code FF puts something like a long dash on the screen. Printing four of these characters side by side draws a line on the screen! To get dashes across the screen you must leave a space by setting two dots in each block "off." To do this, the code is 33.



Notice that, when you stop the program, the center of the screen fills with 16 exclamation points(!).

Now enter some other codes and experiment with the program until you feel comfortable with the shorthand codes. To help you work out the codes, draw up several 8-by-8 grids and mark off your "dots-on, dots-off" desing. Then figure out the code you need for each block of the grid.

### A Block Figure with CALL CHAR

Now that you've had some experience with defining your own characters, let's see if we can create a small "human" figure by turning dots on and off.

To begin, you need to create the figure on a character grid worksheet like the one below. (Later, when you are creating your own characters, you may want to make copies of the worksheet, not only to design your symbols but also to use in translating the symbol into the shorthand code of the CALL CHAR statement.)

CHAR Worksheet

|       | LEFT<br>BLOCK | RIGHT<br>BLOCK | CODE | SHORT<br>HAND<br>CODE | DOTS |
|-------|---------------|----------------|------|-----------------------|------|
| ROW 1 | □             | □              | —    | 0                     | 0000 |
| ROW 2 | □             | □              | —    | 1                     | 0001 |
| ROW 3 | □             | □              | —    | 2                     | 0010 |
| ROW 4 | □             | □              | —    | 3                     | 0011 |
| ROW 5 | □             | □              | —    | 4                     | 0100 |
| ROW 6 | □             | □              | —    | 5                     | 0101 |
| ROW 7 | □             | □              | —    | 6                     | 0110 |
| ROW 8 | □             | □              | —    | 7                     | 0111 |
|       |               |                | —    | 8                     | 1000 |
|       |               |                | —    | 9                     | 1001 |
|       |               |                | —    | A                     | 1010 |
|       |               |                | —    | B                     | 1011 |
|       |               |                | —    | C                     | 1100 |
|       |               |                | —    | D                     | 1101 |
|       |               |                | —    | E                     | 1110 |
|       |               |                | —    | F                     | 1111 |

INPUT TO CHAR: \_\_\_\_\_

Using the worksheet, we'll mark ones (1's) in the positions where the dots will be turned on:

**CHAR Worksheet**

|       | LEFT BLOCK | RIGHT BLOCK | CODE | SHORT HAND CODE | DOTS |
|-------|------------|-------------|------|-----------------|------|
| ROW 1 | 1          | 1 1         | —    | 0               | 0000 |
| ROW 2 | 1          | 1 1         | —    | 1               | 0001 |
| ROW 3 |            | 1 1 1 1     | —    | 2               | 0010 |
| ROW 4 |            | 1 1 1 1     | —    | 3               | 0011 |
| ROW 5 |            | 1 1 1 1     | —    | 4               | 0100 |
| ROW 6 |            | 1 1 1 1     | —    | 5               | 0101 |
| ROW 7 |            | 1 1         | —    | 6               | 0110 |
| ROW 8 |            | 1 1         | —    | 7               | 0111 |
|       |            |             |      | 8               | 1000 |
|       |            |             |      | 9               | 1001 |
|       |            |             |      | A               | 1010 |
|       |            |             |      | B               | 1011 |
|       |            |             |      | C               | 1100 |
|       |            |             |      | D               | 1101 |
|       |            |             |      | E               | 1110 |
|       |            |             |      | F               | 1111 |

INPUT TO CHAR: \_\_\_\_\_

Now, let's look at the same figure with the "on" dots shaded in, and let's fill in the shorthand codes for developing the character. This form of the worksheet shows you what the character will look like on the screen.

**CHAR Worksheet**

|       | LEFT BLOCK | RIGHT BLOCK | CODE | SHORT HAND CODE | DOTS |
|-------|------------|-------------|------|-----------------|------|
| ROW 1 | ■ □ □ □    | ■ □ □ □     | 99   | 0               | 0000 |
| ROW 2 | ■ ■ □ □    | ■ ■ □ □     | 5A   | 1               | 0001 |
| ROW 3 | □ ■ ■ ■    | □ ■ ■ ■     | 3C   | 2               | 0010 |
| ROW 4 | □ ■ ■ ■    | □ ■ ■ ■     | 3C   | 3               | 0011 |
| ROW 5 | □ ■ ■ ■    | □ ■ ■ ■     | 3C   | 4               | 0100 |
| ROW 6 | □ ■ ■ ■    | □ ■ ■ ■     | 3C   | 5               | 0101 |
| ROW 7 | □ ■ □ □    | □ ■ □ □     | 24   | 6               | 0110 |
| ROW 8 | □ ■ □ □    | □ ■ □ □     | 24   | 7               | 0111 |
|       |            |             |      | 8               | 1000 |
|       |            |             |      | 9               | 1001 |
|       |            |             |      | A               | 1010 |
|       |            |             |      | B               | 1011 |
|       |            |             |      | C               | 1100 |
|       |            |             |      | D               | 1101 |
|       |            |             |      | E               | 1110 |
|       |            |             |      | F               | 1111 |

INPUT TO CHAR: 995A3C3C3C2424

By filling in the worksheet for both the character and the shorthand codes, we know that one line of our program will be:

```
LET A$ = "995A3C3C3C2424"
```

But before we actually start our program, we need to discuss a bit further the process of defining a character. In our previous examples we *redefined* an already existing character, the exclamation point (character code 33). There are other character codes, however, that are *undefined* by the computer. These are available for you to use in building a customized character set in your graphics programs. The undefined character codes are grouped into the following sets (for color graphics):

| Set # 9 | Set # 10 | Set # 11 | Set # 12 |
|---------|----------|----------|----------|
| 96      | 104      | 112      | 120      |
| 97      | 105      | 113      | 121      |
| 98      | 106      | 114      | 122      |
| 99      | 107      | 115      | 123      |
| 100     | 108      | 116      | 124      |
| 101     | 109      | 117      | 125      |
| 102     | 110      | 118      | 126      |
| 103     | 111      | 119      | 127      |

| Set # 13 | Set # 14 | Set # 15 | Set # 16 |
|----------|----------|----------|----------|
| 128      | 136      | 144      | 152      |
| 129      | 137      | 145      | 153      |
| 130      | 138      | 146      | 154      |
| 131      | 139      | 147      | 155      |
| 132      | 140      | 148      | 156      |
| 133      | 141      | 149      | 157      |
| 134      | 142      | 150      | 158      |
| 135      | 143      | 151      | 159      |

These codes and their corresponding set numbers are used in the CALL CHAR, CALL HCHAR, CALL VCHAR, and CALL COLOR statements exactly as we used the defined character codes and their set numbers. Let's use code 96 in our sample program.

OK, we're ready to begin our program. Enter these lines:

```

NEW
10 REM**LITTLE PERSON**
20 CALL CLEAR
30 LET A$ = "995A3C3C3C2424"
40 CALL CHAR (96, A$)
50 CALL COLOR (9, 2, 16)
60 CALL VCHAR (12, 16, 96)
70 GO TO 70

```

The shorthand code for our "figure"  
 Define character code 96  
 White  
 Black  
 Set number  
 Display character

Now run the program and observe the small “person” on the screen. Remember, the figure is only one character in size, so look closely. When you’re ready to stop the program, press **SHIFT C**.

Would it be possible to animate our little figure? Yes, it would! By changing our program and incorporating one of the techniques we covered under Animation, we can turn our character into Mr. Bojangles, the dancing man!

As it’s presently written, our program defines only one character. To make Mr. Bojangles appear to move, we’ll need to define two characters that are alternately displayed in the same position. So we’ll go to our CHAR Worksheets to design our two new characters.

#### CHAR Worksheet—First Figure

|       | LEFT<br>BLOCK | RIGHT<br>BLOCK | CODE | SHORT<br>HAND<br>CODE | DOTS |
|-------|---------------|----------------|------|-----------------------|------|
| ROW 1 | 1             | 1              | 99   | 0                     | 0000 |
| ROW 2 | 1             | 1              | 5A   | 1                     | 0001 |
| ROW 3 | 1             | 1              | 3C   | 2                     | 0010 |
| ROW 4 | 1             | 1              | 3C   | 3                     | 0011 |
| ROW 5 | 1             | 1              | 3C   | 4                     | 0100 |
| ROW 6 | 1             | 1              | 3C   | 5                     | 0101 |
| ROW 7 | 1             | 1              | 3C   | 6                     | 0110 |
| ROW 8 | 1             | 1              | 44   | 7                     | 0111 |
|       |               |                | 84   | 8                     | 1000 |
|       |               |                |      | 9                     | 1001 |
|       |               |                |      | A                     | 1010 |
|       |               |                |      | B                     | 1011 |
|       |               |                |      | C                     | 1100 |
|       |               |                |      | D                     | 1101 |
|       |               |                |      | E                     | 1110 |
|       |               |                |      | F                     | 1111 |

INPUT TO CHAR: “995A3C3C3C34484”

#### CHAR Worksheet—Second Figure

|       | LEFT<br>BLOCK | RIGHT<br>BLOCK | CODE | SHORT<br>HAND<br>CODE | DOTS |
|-------|---------------|----------------|------|-----------------------|------|
| ROW 1 |               | 1              | 18   | 0                     | 0000 |
| ROW 2 | 1             | 1              | 99   | 1                     | 0001 |
| ROW 3 | 1             | 1              | FF   | 2                     | 0010 |
| ROW 4 | 1             | 1              | 3C   | 3                     | 0011 |
| ROW 5 | 1             | 1              | 3C   | 4                     | 0100 |
| ROW 6 | 1             | 1              | 3C   | 5                     | 0101 |
| ROW 7 | 1             | 1              | 3C   | 6                     | 0110 |
| ROW 8 | 1             | 1              | 22   | 7                     | 0111 |
|       | 1             | 1              | 21   | 8                     | 1000 |
|       |               |                |      | 9                     | 1001 |
|       |               |                |      | A                     | 1010 |
|       |               |                |      | B                     | 1011 |
|       |               |                |      | C                     | 1100 |
|       |               |                |      | D                     | 1101 |
|       |               |                |      | E                     | 1110 |
|       |               |                |      | F                     | 1111 |

INPUT TO CHAR: “1899FF3C3C3C2221”

Now we're ready to edit the program. Enter these lines:

```

30 A$ = "995A3C3C3C3C4484"
35 B$ = "1899FF3C3C3C2221"
45 CALL CHAR (97, B$)
70 FOR DELAY = 1 TO 100 }
80 NEXT DELAY }
90 CALL VCHAR (12, 16, 97) }
100 FOR DELAY }
110 GO TO 60 }

```

Annotations:

- First character (points to the first '9' in line 30)
- Second character (points to the first 'A' in line 30)
- Define character code 97 as B\$ (points to the 'B\$' in line 45)
- Display first figure (points to the closing brace of the first loop in line 80)
- Display second figure (points to the closing brace of the second loop in line 100)
- Return and display first character, repeating the whole procedure (points to the 'GO TO 60' in line 110)

Clear the screen and list the changed program so that you can see how it fits together:

```

LIST
10 REM**LITTLE PERSON**
20 CALL CLEAR
30 A$ = "995A3C3C3C3C4484"
35 B$ = "1899FF3C3C3C2221"
40 CALL CHAR (96, A$)
50 CALL COLOR (9, 2, 16)
60 CALL VCHAR (12, 16, 96)
70 FOR DELAY = 1 TO 100
80 NEXT DELAY
90 CALL VCHAR (12, 16, 97)
100 FOR DELAY = 1 TO 100
110 NEXT DELAY
120 GO TO 60

```

Now run the program and watch Mr. Bojangles dance! (To stop the program, press **SHIFT C**).

After running the program a few times, you might like to add a FOR-NEXT loop to make Mr. Bojangles dance across the screen. Also, try creating other pairs of characters and placing their shorthand codes in lines 30 and 35. Can you turn Mr. Bojangles into an acrobat who flips from his hands to his feet and back again?

As we've mentioned, Mr. Bojangles is pretty small —only one character in size. Not all the designs you can create are limited to this small size. You can combine several small characters to construct bigger graphics that cover more of the screen. Our next program shows how to design a larger graphic using one small color character as our "building block."

**The Giant**

If you define one special character where all the dots are “on,” you can then use it to paint in the rest of a large figure. The following program takes the small character just mentioned and creates a “giant” figure similar to the Mr. Bojangles character. Enter the program and see what it does:

```

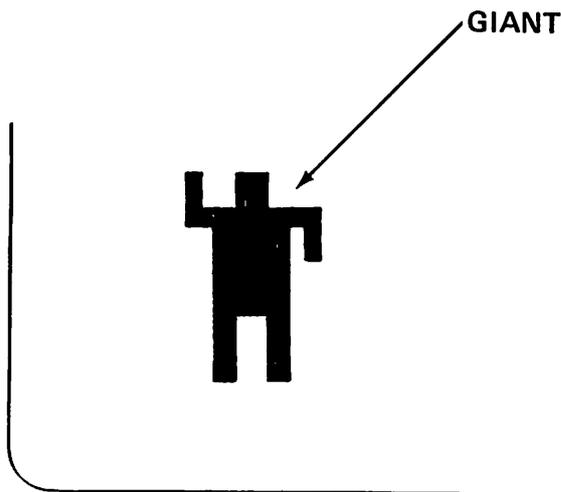
NEW
10 REM**THE GIANT**
20 CALL CLEAR
30 A$ = "FFFFFFFFFFFFFFFF"
40 CALL CHAR (96, A$)
50 CALL COLOR (9, 5, 5)
60 CALL VCHAR (7, 15, 96, 8)
70 CALL VCHAR (7, 16, 96, 8)
80 CALL VCHAR (9, 14, 96, 10)
90 CALL VCHAR (9, 17, 96, 10)
100 CALL VCHAR (7, 12, 96, 3)
110 CALL VCHAR (9, 13, 96)
120 CALL VCHAR (9, 19, 86, 3)
130 CALL VCHAR (9, 18, 96)
140 GO TO 140

```

Annotations:

- Shorthand code to turn on all the dots in the character (points to line 40)
- Set character color – dark blue (foreground and background) (points to line 50)
- “Build” head and center of torso. (points to lines 60-70)
- Left torso and leg (points to line 80)
- Right torso and leg. (points to line 90)
- Left arm (points to line 100)
- Right arm (points to line 110)
- Hold graphic on screen (points to line 140)

When you run the program, you’ll see a larger version of Mr. Bojangles:



Our dark-blue “giant” is rather angular and blocky, since it’s created from a single angular character. You might like to rework the program, adding extra defined characters that allow you to soften the edges of the figure.

Experiment with some other block designs using your “all-dots-on” character. Then try defining other characters to include in your graphics programs. The examples shown below will help to get you started:

CHAR Worksheet

|       | LEFT BLOCK |  |  |  | RIGHT BLOCK |   |   |   | CODE |
|-------|------------|--|--|--|-------------|---|---|---|------|
| ROW 1 |            |  |  |  |             |   |   | 1 | 01   |
| ROW 2 |            |  |  |  |             |   | 1 | 1 | 03   |
| ROW 3 |            |  |  |  |             | 1 | 1 | 1 | 07   |
| ROW 4 |            |  |  |  |             | 1 | 1 | 1 | 0F   |
| ROW 5 |            |  |  |  |             | 1 | 1 | 1 | 1F   |
| ROW 6 |            |  |  |  |             | 1 | 1 | 1 | 3F   |
| ROW 7 |            |  |  |  |             | 1 | 1 | 1 | 7F   |
| ROW 8 |            |  |  |  |             | 1 | 1 | 1 | FF   |

INPUT TO CHAR: “0103070F1F3F7FFF”

CHAR Worksheet

|       | LEFT BLOCK |  |  |  | RIGHT BLOCK |   |   |   | CODE |
|-------|------------|--|--|--|-------------|---|---|---|------|
| ROW 1 |            |  |  |  |             |   | 1 | 1 | 18   |
| ROW 2 |            |  |  |  |             | 1 | 1 | 1 | 3C   |
| ROW 3 |            |  |  |  |             | 1 | 1 | 1 | 7E   |
| ROW 4 |            |  |  |  |             | 1 | 1 | 1 | FF   |
| ROW 5 |            |  |  |  |             | 1 | 1 | 1 | FF   |
| ROW 6 |            |  |  |  |             | 1 | 1 | 1 | 7E   |
| ROW 7 |            |  |  |  |             | 1 | 1 | 1 | 3C   |
| ROW 8 |            |  |  |  |             | 1 | 1 |   | 18   |

INPUT TO CHAR: “183C7EFFFF7E3C18”

CHAR Worksheet

|       | LEFT BLOCK |   |   |   | RIGHT BLOCK |   |   |   | CODE |
|-------|------------|---|---|---|-------------|---|---|---|------|
| ROW 1 | 1          | 1 | 1 | 1 | 1           | 1 | 1 | 1 | FF   |
| ROW 2 |            | 1 | 1 | 1 | 1           | 1 | 1 |   | 7E   |
| ROW 3 |            |   | 1 | 1 | 1           | 1 |   |   | 3C   |
| ROW 4 |            |   |   | 1 | 1           |   |   |   | FF   |
| ROW 5 |            |   |   |   | 1           | 1 |   |   | FF   |
| ROW 6 |            |   |   |   | 1           | 1 | 1 |   | 3C   |
| ROW 7 |            |   |   |   | 1           | 1 | 1 | 1 | 7E   |
| ROW 8 | 1          | 1 | 1 | 1 | 1           | 1 | 1 | 1 | FF   |

INPUT TO CHAR: “FF7E3C18183C7EFF”

**Summary of Chapter Eleven**

This chapter covered a lot of material on the use of color, sound, and graphics. Yet you were introduced to just three new TI BASIC features:

- CALL SCREEN – Changes the color of the screen.
- CALL CHAR – Creates new graphics characters
- CALL KEY – Inputs character codes from the keyboard.

However, you were shown many important techniques for handling and creating exciting visual and auditory accents for programs.

You may find that you will often refer back to the information in this chapter as you begin to create your own programs. Some of the principal differences between older computers and machines such as the TI Home Computer are in the expanded use of color graphics and sounds. Even beginning programmers can now use these features.

Go back and spend some more time in this chapter, if you like. Change the programs, as suggested, or make up your own experiments. The mastery of the color, sound, and graphics of the TI Home Computer may well be one of the more rewarding experiences you have had in computing.

When you are ready, proceed onward to the exercises on this material. Enjoy!

## Chapter Eleven Exercises

- (1) The color codes for black and dark red are four (4) and seven (7) respectively. What happens when the following statements are executed within a program?

(a) **CALL SCREEN (7)** \_\_\_\_\_  
 \_\_\_\_\_  
 \_\_\_\_\_

(b) **CALL SCREEN (4)** \_\_\_\_\_  
 \_\_\_\_\_  
 \_\_\_\_\_

- (2) Describe what the two parameters in **CALL CHAR** are used for?

**CALL CHAR (C, A\$)**

↙ ↘

(a) \_\_\_\_\_ (b) \_\_\_\_\_

- (3) Assume that **A\$** contains a string of characters that will cause **CALL CHAR** to redefine the asterisk to be the symbol:



Draw a picture of the block of characters that appear near the center of the screen when the following lines of a program are executed:

```
100 CALL CHAR (42, A$)
110 FOR COUNT = 1 TO 5
120 CALL VCHAR (12+COUNT, 16, 42, COUNT)
130 NEXT COUNT
```

- (4) Draw a picture of the pattern that appears on the screen when the following program is executed. (Again, assume **A\$** contains a string that will produce the symbol  when used with **CALL CHAR**.)

```
100 CALL CHAR (42, A$)
110 FOR ROW = 1 TO 5
120 FOR COL = 1 TO ROW
130 PRINT "*" ;
140 NEXT COL
150 PRINT
160 NEXT ROW
```

- (5) You guessed it! This exercise asks you to define the string of characters that go into **A\$** that produces the symbol , used in the last two exercises.

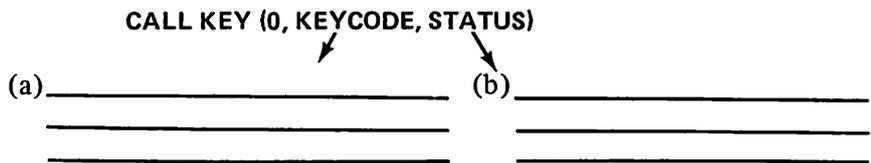
**A\$ = " \_\_\_\_\_ "**

- (6) Draw the picture for the symbol that is produced by the following:

```
10 A$ = "7EA5819981BD817E"
20 CALL CHAR (42, A$)
30 CALL VCHAR (12, 16, 42)
```

|       | LEFT<br>BLOCK |  |  |  | RIGHT<br>BLOCK |  |  |  |
|-------|---------------|--|--|--|----------------|--|--|--|
| ROW 1 |               |  |  |  |                |  |  |  |
| ROW 2 |               |  |  |  |                |  |  |  |
| ROW 3 |               |  |  |  |                |  |  |  |
| ROW 4 |               |  |  |  |                |  |  |  |
| ROW 5 |               |  |  |  |                |  |  |  |
| ROW 6 |               |  |  |  |                |  |  |  |
| ROW 7 |               |  |  |  |                |  |  |  |
| ROW 8 |               |  |  |  |                |  |  |  |

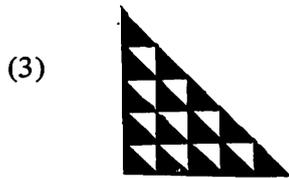
- (7) The CALL KEY function has three parameters. The first is a zero. Describe the second and third parameters:



- (8) Create a program that does the following:
- Redefines the asterisk to be the symbol **■** (a solid block).
  - Puts the symbol on the left edge of the screen, and then causes it to "move" across the screen to the right side.
  - Keeps repeating (b).
- (9) Add a CALL SOUND to the program in Exercise 8 that causes a sound or noise to be made when the symbol hits the right side of the screen.
- (10) Change the program in Exercise 8 so that a sound or noise is made each time the symbol "moves." Have the sound increase in frequency with each "move."

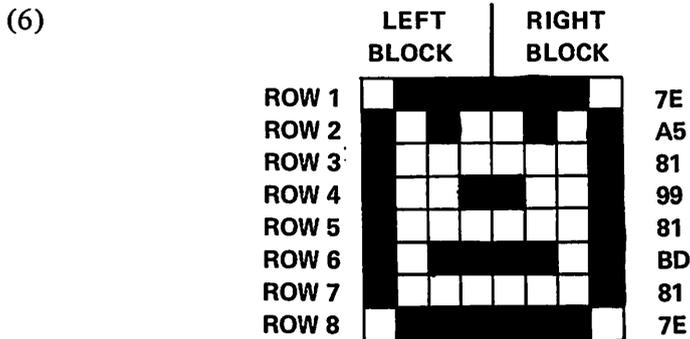
Answers to Chapter Eleven Exercises

- (1) (a) The background screen color turns dark red.
- (b) The background screen color turns black. Since the message and characters during Run Mode are black, they will disappear.
- (2) (a) The first parameter is the character code for the character that is being redefined.
- (b) The second parameter , a string variable, contains the code for the new symbol that is replacing the character being redefined.



(4) The picture looks the same as that in exercise 3.

(5) A\$ = "80C0E0F0F8CFE0FF"



- (7) (a) The second parameter will contain the character code of the key that is depressed on the keyboard. If an asterisk key is depressed, KEYCODE would contain 42.
- (b) The third parameter is the status indicator. STATUS is zero if no key is depressed; one if a new key was pressed since the last time CALL KEY was executed; minus one (-1) if the same key as last time was depressed.

(8) Here is a program for Exercise 8. Your program may be different!

```

10 REM**MOVING SYMBOL PROGRAM**
20 A$ = "FFFFFFFFFFFFFFFF"
30 CALL CHAR (42, A$)
40 FOR COL = 2 TO 28
50 CALL VCHAR (12, COL-1, 32)
60 CALL VCHAR (12, COL, 42)
70 NEXT COL
80 CALL VCHAR (12, 28, 32)
90 GO TO 40

```

Redefine \*\* as ■  
 Begin loop over columns  
 Blank last space to left  
 Put symbol on screen  
 Blank last position  
 Repeat it all

- (9) Add the following lines to make a sound at the end of the line each time. (Your line numbers and sound parameters may be different.)

```
75 CALL SOUND (100, 220, 2)
```

- (10) To make a sound each time the symbol “moves,” and have the sound increase in frequency each time, make the following changes to the program in Exercise 8. (You may have different changes.)

```
35 NOTE = 220
```

```
65 CALL SOUND (100, NOTE, 2)
```

```
67 NOTE = NOTE + 20
```

```
90 GO TO 35
```

# Chapter Twelve

## More Strings

You have used programs that contained strings in previous chapters. They were used in INPUT, PRINT, DATA, assignment, and comparison statements. Typical examples are:

```
INPUT "COLOR ?": X
PRINT "X = "; X
A$ = "CHARACTER"
DATA "CHECK", "DEPOSIT"
IF A$ = "YES" THEN 600
```

In this chapter, you will learn to use new statements that allow you to search existing strings. You will learn to:

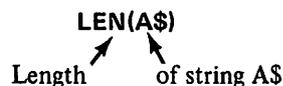
- Find the length (the number of characters) of a string.
- Extract a substring from a string.
- Find the ASCII numeric value of a character contained in a string.
- Print an ASCII character corresponding to a numeric value.
- Find the position of a given character in a string.
- Convert a numeric expression or constant to a string.
- Convert a string to a numeric expression or constant.
- Concatenate strings.

### The Length of a String

The length of a string may be found by using the statement:

LEN(A\$)

Length      of string A\$



Example:

```
A$ = "THIS STRING HAS 30 CHARACTERS."
 ↑ ↑ ↑ ↑ ↑ ↑
 5 10 15 20 25 30
```

All characters and spaces within the quotation marks are counted. Punctuation marks within the quotation marks are counted. The quotation marks at the beginning and end of the string are *not* counted.

Enter and run the following program.

```
100 CALL CLEAR
110 A$ = "THIS STRING HAS 30 CHARACTERS."
120 PRINT A$
130 PRINT LEN(A$)
```

When the run has been completed, you will see:

```
THIS STRING HAS 30 CHARACTERS. ← A$
30 ← LEN(A$)
DONE
> □
```

The LEN statement may also be used to compare the length of two strings. The following program compares the length of A\$ and B\$. It then prints the results.

```
100 CALL CLEAR
110 REM**HERE ARE TWO STRINGS**
120 A$ = "THIS STRING HAS 30 CHARACTERS."
130 B$ = "THIS ONE HAS 27 CHARACTERS."
140 REM**COMPARE THEIR LENGTHS**
150 IF LEN(A$)>LEN(B$) THEN 190
160 REM**PRINT RESULTS**
170 PRINT "B$ IS LONGER THAN A$."
180 GO TO 200
190 PRINT "A$ IS LONGER THAN B$."
```

Of course A\$ is longer than B\$ in the program above. The printed message will always be the same.

Now it's your turn to input two strings. The computer will tell you which is longer. It then goes back to the beginning of the program to let you input two more strings.

```

100 CALL CLEAR
110 REM**INPUT FIRST STRING**
120 PRINT "TYPE THE FIRST STRING."
130 INPUT A$
140 REM**INPUT SECOND STRING**
150 PRINT "TYPE THE SECOND STRING."
160 INPUT B$
170 REM**COMPARE LENGTHS**
180 IF LEN(A$)>LEN(B$) THEN 230
190 IF LEN(A$) = LEN(B$) THEN 250
200 REM**PRINT RESULTS**
210 PRINT "SECOND STRING IS LONGER."
220 GO TO 110
230 PRINT "FIRST STRING IS LONGER."
240 GO TO 110
250 PRINT "THE STRINGS ARE EQUAL."
260 GO TO 110

```

When you run the program, you will find out which string is longer than the other. But, you won't find out how much longer it is. Let's change lines 210 and 230 to provide this information.

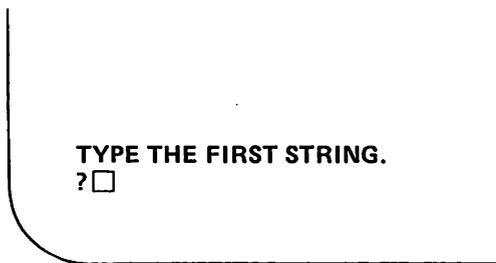
Change:

```

210 PRINT "THE SECOND IS ";LEN(B$)-LEN(A$),"LONGER
 THAN THE FIRST."
230 PRINT "THE FIRST IS ";LEN(A$)-LEN(B$),"LONGER
 THAN THE SECOND."

```

When you change these two lines and run the program again, the computer asks for the first input.



```

TYPE THE FIRST STRING.
?□

```

Type in whatever you want. We typed:

```

THIS IS OUR FIRST STRING.

```

The computer then asks for the second string.

```

TYPE THE FIRST STRING.
? THIS IS OUR FIRST STRING.
TYPE THE SECOND STRING.
? □

```

Now you type in a second message. We typed:

**THIS IS OUR SECOND STRING.**

and then pressed **ENTER**. The computer then compares the lengths of the messages and prints the results. This is our result.

```

TYPE THE FIRST STRING.
? THIS IS OUR FIRST STRING.
TYPE THE SECOND STRING.
? THIS IS OUR SECOND STRING.
SECOND STRING IS 1
LONGER THAN THE FIRST. } Result
TYPE THE FIRST STRING. }
? □ ← Asks for another

```

The computer is now ready to accept the next pair of strings for comparison. Experiment with several pairs of strings to assure yourself that each of the three responses will be printed correctly.

### Selecting a Substring of a String

Suppose that you have a string consisting of:

**A\$ = "ONE TWO THREE FOUR FIVE"**

You want to pick out one of the words (a substring) from the string. To do this you can use the statement:

**SEG\$(A\$,X,Y)**

A\$ is the name of the original string. X is the numeric value of the position (numbered from left to right) of the first character of the substring that you wish to pick out. Y is the number of characters in the desired substring.

If you wanted to print the first word in the string, you would use the statement:

```
PRINT SEG$(A$,1,3)
 ↑ ↑
 | | Print three characters
 | | Start from 1st position
```

This would print the word: ONE

If you wanted to print the second word in the string, you would use the statement:

```
PRINT SEG$(A$,5,3)
 ↑ ↑
 | | Print three characters
 | | Start from 5th position
```

This would print the word: TWO

If you wanted to print the last word in the string, you would use the statement:

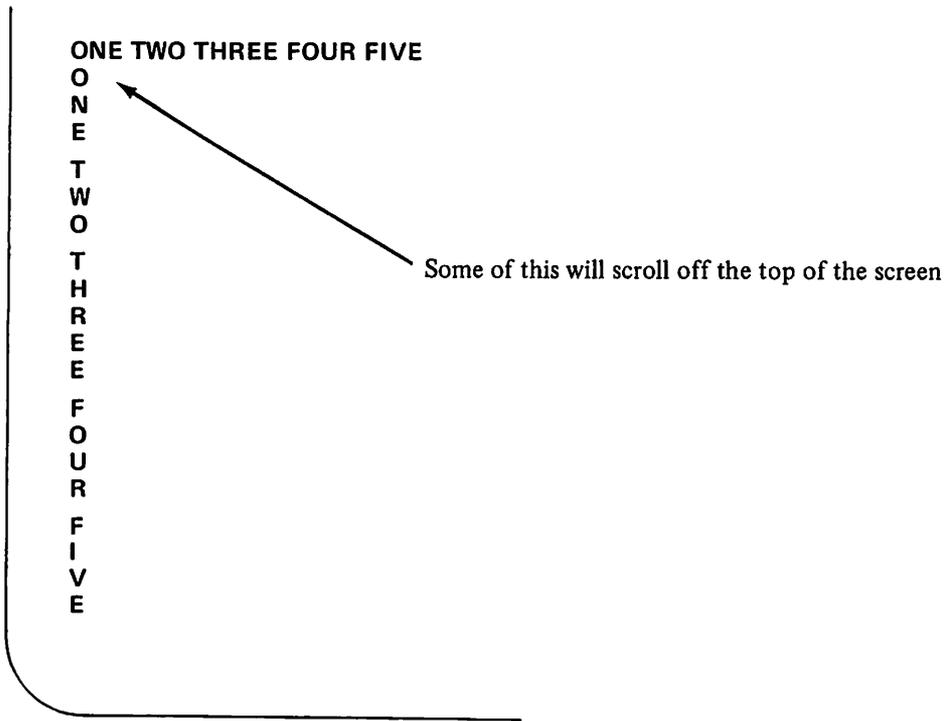
```
PRINT SEG$(A$,20,4)
 ↑ ↑
 | | Print four characters
 | | Start from 20th position
```

This would print the word: FIVE

Using the SEG\$ statement, you can actually pick out one letter of the string at a time. Enter and run the following program. It will pick out and print each letter from the original string. Try it.

```
100 CALL CLEAR
110 REM**ASSIGN AND PRINT STRING**
120 A$ = "ONE TWO THREE FOUR FIVE"
130 PRINT A$
140 REM**PRINT EACH CHARACTER**
150 FOR X = 1 TO 23
160 PRINT SEG$(A$,X,1)
170 NEXT X
180 GO TO 180
 ↑ ↑
 | | Print only one character
 | | Start with character number X
```

The run will print this:



Notice that the spaces between words were also shown. These count as characters in the SEG\$ statement.

Here is a program that combines LEN and SEG statements. Substrings are picked out, and their lengths are compared.

```
100 CALL CLEAR
110 REM**ASSIGN STRING**
115 PRINT "HERE IS THE STRING."
120 A$ = "ONE TWO THREE FOUR FIVE"
125 PRINT A$

130 REM**CHOOSE FIRST SUBSTRING**
140 INPUT "FIRST WORD (1-5)? " : B

150 REM**CHECK FIRST SUBSTRING**
160 ON B GOSUB 500,520,540,560,580

170 REM**CHOOSE SECOND SUBSTRING**
180 INPUT "SECOND WORD (1-5)? " : C

190 REM**CHECK SECOND SUBSTRING**
200 ON C GOSUB 600,620,640,660,680

210 REM**COMPARE SUBSTRINGS**
220 IF LEN(B$) > LEN(C$) THEN 270
230 IF LEN(B$) = LEN(C$) THEN 290

240 REM**PRINT RESULTS**
250 PRINT C$;" IS LONGER THAN " ; B$
260 GO TO 300
270 PRINT B$;" IS LONGER THAN " ; C$
280 GO TO 300
290 PRINT "LENGTHS OF ";B$ " AND ";C$;" ARE EQUAL"

300 END

500 REM**LABEL THE SUBSTRINGS**
505 B$ = SEG$(A$,1,3)
510 RETURN

520 B$ = SEG$(A$,5,3)
530 RETURN

540 B$ = SEG$(A$,9,5)
550 RETURN

560 B$ = SEG$(A$,15,4)
570 RETURN

580 B$ = SEG$(A$,20,4)
590 RETURN

600 C$ = SEG$(A$,1,3)
610 RETURN

620 C$ = SEG$(A$,5,3)
630 RETURN

640 C$ = SEG$(A$,9,5)
650 RETURN

660 C$ = SEG$(A$,15,4)
670 RETURN

680 C$ = SEG$(A$,20,4)
690 RETURN
```

## How to Use the Program

The computer first prints the string:

**ONE TWO THREE FOUR FIVE**

It then asks you to select the first word by inputting the *number* of the word.

1 for the word ONE

or

2 for the word TWO

or

3 for the word THREE

or

4 for the word FOUR

or

5 for the word FIVE

It then asks for the second word. Once again type in one of the numbers 1,2,3,4, or 5. The computer then compares the lengths of the two words. It prints each word and tells which is the larger.

Examples:

```

HERE IS THE STRING.
ONE TWO THREE FOUR FIVE
FIRST WORD (1-5)? 4
SECOND WORD (1-5)? 5
LENGTHS OF FOUR AND FIVE
ARE EQUAL
DONE
>□

```

```

HERE IS THE STRING.
ONE TWO THREE FOUR FIVE
FIRST WORD (1-5)? 1
SECOND WORD (1-5)? 3
THREE IS LONGER THAN ONE
DONE
>□

```

```

HERE IS THE STRING.
ONE TWO THREE FOUR FIVE
FIRST WORD (1-5)? 5
SECOND WORD (1-5)? 2
FIVE IS LONGER THAN TWO
DONE
>□

```



Here's the result.

```

STRING FUN
DONE
>□

```

Let's play some more. Change lines 140 and 150 to:

```

140 A$ = SEG$(A$,15,7) & " "
150 B$ = SEG$(Z$,1,8)

```

Blank space

Note in line 140: Quoted strings may be concatenated with substrings. The run shows:

```

STRINGS HAVE FUN
DONE
>□

```

See if you can figure out what the display will show if lines 140 and 150 are changed as shown below. Then make the changes and rerun the program.

```

140 A$ = SEG$(A$,1,14)
150 B$ = SEG$(Z$,11,2)

```

The run:

```

HAVE FUN WITH IT
DONE
>□

```

The next one is a little more difficult. Change the lines to:

```

140 A$ = SEG$(Z$,17,4) & SEG$(Z$,18,3)
150 B$ = SEG$(Z$,14,8)

```

The run:

```

RINGING STRINGS
DONE
>□

```

One last change:

```

140 A$ = SEG$(Z$,11,2) & " " & SEG$(Z$,1,2) & SEG$(Z$,15,1)
150 B$ = SEG$(Z$,5,2) & SEG$(Z$,11,2) & SEG$(Z$,15,1)

```

The run:

```

IT HAS FITS
DONE
>□

```

Here is a program that will let you input your own string, pick out three substrings, and concatenate the substrings into a new string. If you have only two substrings, input any positive number for E (the starting character of the third substring) and input 0 (zero) for F. This will return a null string (nothing) for the third string.

#### PLAYING ON THE STRINGS

```

100 CALL CLEAR
110 REM**INPUT THE STRING**
120 PRINT "WHAT IS YOUR STRING"
130 INPUT Z$
140 REM**CHOOSE SUBSTRINGS**
150 PRINT "STARTING CHARACTER","FIRST SUBSTRING"
160 INPUT A
170 PRINT "NUMBER OF CHARACTERS"
180 INPUT B
190 PRINT "STARTING CHARACTER","SECOND SUBSTRING"
200 INPUT C
210 PRINT "NUMBER OF CHARACTERS"
220 INPUT D
230 PRINT "STARTING CHARACTER","THIRD SUBSTRING"
240 INPUT E
250 PRINT "NUMBER OF CHARACTERS"
260 INPUT F
270 REM**FIND SUBSTRINGS**
280 A$ = SEG$(Z$,A,B)
290 B$ = SEG$(Z$,C,D)
300 C$ = SEG$(Z$,E,F)
310 REM**PRINT NEW STRING**
320 CALL CLEAR
330 PRINT A$ & B$ & C$
340 PRINT
350 GO TO 110

```

Sample run – first input the data:

```

WHAT IS YOUR STRING
? SLOWLY SLIPPING STRINGS
STARTING CHARACTER
FIRST SUBSTRING
? 12
NUMBER OF CHARACTERS
? 4
STARTING CHARACTER
SECOND SUBSTRING
? 13
NUMBER OF CHARACTERS
? 3
STARTING CHARACTER
THIRD SUBSTRING
? 16
NUMBER OF CHARACTERS
? 8

```

The screen is then cleared to print the results.

```

PINGING STRINGS ← Rearranged string
WHAT IS YOUR STRING ← Ready for a new string
?

```

Experiment with this program until you have a good feel for manipulating strings. When you have finished, we'll look at some more string statements.

## ASCII Codes

The computer uses ASCII codes to convert numbers to a form that will produce alphanumeric characters on the screen. BASIC has a statement that will display the ASCII code for any alphanumeric and other characters.

For example:

```
PRINT ASC("A")
```

would display:

```
65 ← The ASCII code for A
```

The statement `ASC(A$)` will return the numeric value of the *first* character of the string, A\$. If

```
A$ = "HAVE FUN WITH STRINGS"
```

the statement: `PRINT ASC(A$)` would display: **72** ← The ASCII code for H

The ASC statement can also be used with the SEG\$ statement to pick any desired character out of a string. The following program illustrates this use.

```
100 CALL CLEAR
110 Z$ = "HAVE FUN WITH STRINGS"
120 PRINT Z$
130 INPUT "WHICH CHARACTER?":A
140 B$ = SEG$(Z$,A,1)
150 PRINT ASC(B$)
```

The program:

- (1) Prints : HAVE FUN WITH STRINGS
- (2) Asks: WHICH CHARACTER?
- (3) You then respond with any integer from 1 through 21.
- (4) The computer responds with the ASCII code for your selected number.

Here are some examples:

```
HAVE FUN WITH STRINGS ← The string
WHICH CHARACTER? 6 ← 6th character is F
70 ← ASCII code for F
DONE
>□
```

```
HAVE FUN WITH STRINGS
WHICH CHARACTER? 10 ← 10th character is W
87 ← ASCII code for W
DONE
>□
```

```
HAVE FUN WITH STRINGS
WHICH CHARACTER? 15 ← 15th character is S
83 ← ASCII code for S
DONE
>□
```

We can print a table for the ASCII codes for the complete alphabet with the following program.

**ASCII-ALPHABET PROGRAM**

```

100 CALL CLEAR
110 REM**PRINT ASCII CODES FOR A-Z**
120 Z$ = "ABCDEFGHIJKLMNOPQRSTUVWXYZ" ← The string
130 FOR W = 1 TO 26
140 A$ = SEG$(Z$,W,1) ← Pick one character starting at position W
150 PRINT A$;ASC(A$),
160 NEXT W
170 END

```

The RUN displays:

```

A 65 B 66
C 67 D 68
E 69 F 70
G 71 H 72
I 73 J 74
K 75 L 76
M 77 N 78
O 79 P 80
Q 81 R 82
S 83 T 84
U 85 V 86
W 87 X 88
Y 89 Z 90

DONE
>□

```

Numerals are also assigned ASCII codes (a complete table of ASCII codes is given in the Appendix). If we modify lines 120 and 130 of the previous program, we can display the ASCII codes of the numerals 0 through 9. Each digit has a separate code.

```

120 Z$ = "0123456789"
130 FOR W = 1 TO 10

```

The run displays the numerals and their ASCII codes.

```

0 48 1 49
2 50 3 51
4 52 5 53
6 54 7 55
8 56 9 57

DONE
>□

```

Remember, ASC(A\$) returns the ASCII code for only the *first* character of the string, A\$. Therefore, we used the SEG\$ statement to move from character to character within the string, Z\$.

### Finding a Character from its ASCII Code

You can also reverse the procedure of the ASC statement. To print an alphanumeric character from its code, you use:

**PRINT CHR\$(XX)** ← Where XX is the ASCII code for the desired character

Examples:

**PRINT CHR\$(55)**

would print: **7** ← The character whose ASCII code is 55

**PRINT CHR\$(65)**

would print: **A** ← The character whose ASCII code is 65

**PRINT CHR\$(87)**

would print: **W** ← The character whose ASCII code is 87

**PRINT CHR\$(32)**

would print: blank ← The character whose ASCII code is 32

A message can be printed by using CHR\$ to print characters from their ASCII codes. Here is a program that prints such a message.

#### CHR\$ PRINTING PROGRAM

100 CALL CLEAR

110 PRINT CHR\$(84);CHR\$(72);CHR\$(69)

← Prints: THE

120 PRINT CHR\$(65);CHR\$(83);CHR\$(67);CHR\$(73);CHR\$(73)

← Prints: ASCII

130 PRINT CHR\$(67);CHR\$(79);CHR\$(68);CHR\$(69)

← Prints: CODE

140 PRINT CHR\$(70);CHR\$(79);CHR\$(82)

← Prints: FOR

150 PRINT CHR\$(55);CHR\$(32);CHR\$(73);CHR\$(83)

← Prints: 7 IS

160 PRINT CHR\$(53);CHR\$(53)

← Prints: 55

170 END

The run:

```

THE
ASCII
CODE
FOR
7 IS
55
DONE

```

□

## Searching a String

Your computer also has the ability to “look” through a given string in search of an entire second string. Suppose that you have two strings, A\$ and B\$. You can start a search of A\$ at any position within A\$ for the first occurrence of the string B\$. The format for the statement is:

$\text{POS}(\text{A}\$, \text{B}\$, \text{X})$  ← where X is an integer

This statement would find the *first* occurrence of the B\$ in A\$. It would start looking from the Xth position in A\$. For example, if A\$ = “THIS IS A STRING TO BE SEARCHED” and B\$ = “IS”, then

(1)  $\text{POS}(\text{A}\$, \text{B}\$, 1) = 3$

Look for B\$      Start at 1st position of A\$      Found at position 3

B\$ was found at the 3rd position of A\$ for a search that started at position 1.

(2)  $\text{POS}(\text{A}\$, \text{B}\$, 4) = 6$

Start at 4th position this time      Found at position 6

If we start the search at position 4, B\$ is found in position 6.

## Ordering Words

You now have the tools necessary for ordering words according to their alphabetic position. Consider the words:

MONKEY  
ZEBRA  
BEAR

The words as shown are *not* in alphabetical order. Alphabetically arranged, their order would be:

|        |                              |
|--------|------------------------------|
| BEAR   | B comes before M and Z       |
| MONKEY | M comes after B but before Z |
| ZEBRA  | Z comes last                 |

If the names were assigned to string variables, we could use the ASC statement to find the ASCII numeric value for the first character of each string.

Example:

```
A$ = "MONKEY
B$ = "ZEBRA"
C$ = "BEAR"

A = ASC(A$) ← (77)
B = ASC(B$) ← (90)
C = ASC(C$) ← (66)
```

It now becomes easy to put the values of the variables A, B, and C in numerical order. Thus, we have a way to order the words MONKEY, ZEBRA, and BEAR. The program that follows will let you input three strings. It tests only for the first character in each string. If two strings start with the same character, their order will not be switched. Therefore, the inputs should all start with a different letter.

#### ORDERING PROGRAM

```
100 CALL CLEAR
110 REM**INPUT 3 STRINGS**
120 PRINT "FIRST STRING";
130 INPUT A$
140 PRINT "SECOND STRING";
150 INPUT B$
160 PRINT "THIRD STRING";
170 INPUT C$

200 REM**TEST STRINGS AND PRINT**
210 IF ASC(A$) > ASC(B$) THEN 300
220 IF ASC(A$) > ASC(C$) THEN 400
230 PRINT A$
240 IF ASC(B$) > ASC(C$) THEN 500
250 PRINT B$
260 PRINT C$
270 END

300 REM**SWITCH STRINGS 1 AND 2
310 Z$ = B$
320 B$ = A$
330 A$ = Z$
340 GO TO 220

400 REM**SWITCH STRINGS 1 AND 3**
410 Z$ = C$
420 C$ = A$
430 A$ = Z$
440 GO TO 230

500 REM**SWITCH STRINGS 2 AND 3**
510 Z$ = C$
520 C$ = B$
530 B$ = Z$
540 GO TO 250
```

Sample runs:

```

FIRST STRING? MONKEY
SECOND STRING? ZEBRA
THIRD STRING? BEAR
BEAR
MONKEY
ZEBRA
DONE
>

```

```

FIRST STRING? QUEEN
SECOND STRING? JACK
THIRD STRING? KING
JACK
KING
QUEEN
DONE
>

```

### Comparing Two Strings

Let's now look more closely at string comparisons. We can compare more than just the beginning character of two strings by utilizing the `SEG$` statement. Suppose that we want to compare the two words `DODGER` and `DODGEM`. Each word has six letters, and the two words can be compared one letter at a time. A search to produce the ASCII codes for a six letter word could be performed by this loop.

```

100 CALL CLEAR
110 A$ = "DODGER"
120 FOR X = 1 TO 6
130 PRINT ASC(SEG$(A$,X,1))
140 NEXT X

```

This loop would print:

```

68
79
68
71
69
82
DONE
>

```

Now we expand that loop to compare the two words DODGER and DODGEM.

```

100 CALL CLEAR
110 A$ = "DODGER"
120 B$ = "DODGEM"
130 FOR X = 1 TO 6
140 A = ASC(SEG$(A$,X,1))
150 B = ASC(SEG$(B$,X,1))
160 IF A < >
170 NEXT X
180 PRINT "THE WORDS ARE EQUAL"
190 GO TO 240

200 IF A > B THEN 230
210 PRINT A$; " COMES BEFORE "; B$
220 GO TO 240

230 PRINT B$; " COMES BEFORE "; A$
240 END

```

Here is a trace of the program showing the order in which the lines are executed and the value X, A, and B when they are changed.

| Line Executed | X | A  | B  | Remarks   |
|---------------|---|----|----|-----------|
| 100           | — | —  | —  |           |
| 110           | — | —  | —  |           |
| 120           | — | —  | —  |           |
| 130           | 1 | —  | —  |           |
| 140           |   | 68 | —  |           |
| 150           |   |    | 68 |           |
| 160           |   |    |    | D = D     |
| 170           | 2 |    |    | NEXT X    |
| 140           |   | 79 |    |           |
| 150           |   |    | 79 |           |
| 160           |   |    |    | 0 = 0     |
| 170           | 3 |    |    | NEXT X    |
| 140           |   | 68 |    |           |
| 150           |   |    | 68 |           |
| 160           |   |    |    | D = D     |
| 170           | 4 |    |    | NEXT X    |
| 140           |   | 71 |    |           |
| 150           |   |    | 71 |           |
| 160           |   |    |    | G = G     |
| 170           | 5 |    |    | NEXT X    |
| 140           |   | 69 |    |           |
| 150           |   |    | 69 |           |
| 160           |   |    |    | E = E     |
| 170           | 6 |    |    | NEXT X    |
| 140           |   | 82 |    |           |
| 150           |   |    | 83 |           |
| 160           |   |    |    | R < > M   |
|               |   |    |    | GO TO 200 |
| 200           |   |    |    | R < M     |

**210 prints: DODGER COMES BEFORE DODGEM**

If you want to input your own strings, change lines 110 and 120 to:

```
110 INPUT "FIRST WORD?": A$
120 INPUT "SECOND WORD?": B$
```

Sample runs produced by this modification:

```

FIRST WORD? JUMPS
SECOND WORD? JUMPED
JUMPED COMES BEFORE JUMPS ← Since 69<83 (E)<(S)
DONE
>

```

```

FIRST WORD? SEVEN
SECOND WORD? SEVEN
THE WORDS ARE EQUAL ← Since all letters are the same
DONE
>

```

```

FIRST WORD? FELL
SECOND WORD? FELLEDD
FELL COMES BEFORE FELLEDD ← The first four letters are equal. When the fifth
 letter of FELL is not found, SEG$(A$,5,1) is
 assigned a null whose ASCII code is 0. Since
 0<69, FELL comes before FELLEDD.
DONE
>

```

### Numbers to Strings and Strings to Numbers

It is sometimes convenient to use a numeric value as a string. Since computers handle the two types separately, it becomes necessary to convert from one type to the other. To change a numeric expression (X) to a string, you would use the statement:

```
A$ = STR$(X)
```

The number would then be treated as a string.

If X = 17.3, then STR\$(X) equals the string " 17.3".

While arithmetic operations may be performed on the value of X, 17.3, only string operations and functions may be performed on the string " 17.3".

If X = 29.3 and Y = -17.5, then STR\$(X+Y) equals the string " 11.8".

Sample program:

```

100 CALL CLEAR
110 X = 29.3
120 Y = -17.5
130 PRINT "X= ";X
140 PRINT "Y= ";Y
150 PRINT STR$(X)
160 PRINT STR$(Y)
170 PRINT STR$(X + Y)
180 PRINT STR$(X) & STR$(Y)

```

Arithmetic operation on numeric values

String operation on strings

The run:

```

X= 29.3
Y= -17.5
29.3 ← STR$(X)
-17.5 ← STR$(Y)
11.8 ← STR$(X+Y)
29.3-17.5 ← STR$(X) & STR$(Y)
DONE
>□

```

The VAL(A\$) function performs the inverse of the STR\$(X) function. It changes the string A\$ to a numeric constant (provided A\$ is a valid representation of a numeric constant). For example:

- If A\$ = "100", then VAL(A\$) equals the numeric value 100.
- If A\$ = "100", and B\$ = "5", then VAL(A\$ & "." & B\$) equals the numeric value 100.5.

Sample program:

```

100 CALL CLEAR
110 A$ = "100"
120 B$ = "5"
130 PRINT A$
140 PRINT B$
150 PRINT A$ & B$
160 PRINT VAL(A$)
170 PRINT VAL(B$)
180 PRINT VAL(A$ & "." & B$)
190 PRINT VAL(A$) + VAL(B$)

```

String operations on strings

Arithmetic operation on numeric constants

The run:

```

100 ← A$
5 ← B$
1005 ← A$ & B$
100 ← VAL(A$)
5 ← VAL(B$)
100.5 ← VAL(A$ & "." & B$)
105 ← VAL(A$) + VAL(B$)
DONE
>□

```

**Summary of Chapter Twelve**

This chapter has been devoted entirely to strings. You have learned to:

- Find the length of a string.
- Compare the lengths of two strings.
- Select a substring from a string.
- Concatenate substrings of a string.
- Find the ASCII code of a character.
- Find the character of a valid ASCII code.
- Find the relative position of a character within a string.
- Arrange words in alphabetical order.
- Change strings to numeric constant format.
- Change numeric expressions to string format.

## Chapter Twelve Exercises

- (1) If A\$ = "A STRING", what would be the value of LEN(A\$)? \_\_\_\_\_
- (2) Complete the following program to compare the lengths of two strings. Choose your answers from the expressions:

**LONGER THAN, SHORTER THAN, or EQUAL TO**

```

100 INPUT A$
110 INPUT B$
120 IF LEN(A$) < LEN(B$) GO TO 160
130 IF LEN(A$) = LEN(B$) GO TO 180
140 PRINT "A$ IS _____ B$"
150 GO TO 110
160 PRINT "A$ IS _____ B$"
170 GO TO 110
180 PRINT "A$ IS _____ B$"
190 GO TO 110

```

- (3) IF A\$ = "WHO'S FIRST" and B\$ = "WHO'S SECOND", what is the value of LEN(B\$) - LEN(A\$)?
- \_\_\_\_\_

- (4) IF A\$ = "I HOPE YOU GET THIS RIGHT.", what is SEG\$(A\$,21,6)?
- \_\_\_\_\_

- (5) What would this program display when run?

```

100 CALL CLEAR
110 A$ = "PICK A WORD FROM ME."
120 B$ = "THIS STRING IS LONGER."
130 PRINT SEG$(A$,1,7); SEG$(B$,6,6); SEG$(B$,22,1)
140 END

```

- (6) Show what would be on the display after this program in run.

```

100 CALL CLEAR
110 Z$ = "SCRAMBLE THE WORDS"
120 PRINT SEG$(Z$,10,4) & SEG$(Z$,3,6)
130 END

```

(7) What values would be printed by this program?

```

100 CALL CLEAR
110 A$ = "ABC"
120 FOR X = 1 TO 3
130 PRINT ASC(SEG$(A$,X,1))
140 NEXT X
150 END

```

Printed values = \_\_\_\_\_  
 \_\_\_\_\_  
 \_\_\_\_\_

(8) Give the characters for each of these ASCII codes.

CHR\$(82) = \_\_\_\_\_

CHR\$(50) = \_\_\_\_\_

CHR\$(68) = \_\_\_\_\_

CHR\$(84) = \_\_\_\_\_

CHR\$(87) = \_\_\_\_\_

CHR\$(79) = \_\_\_\_\_

(9) IF A\$ = "225" and B\$ = "16",

(a) VAL(A\$ & "." & B\$) = \_\_\_\_\_

(b) VAL(A\$) + VAL(B\$) = \_\_\_\_\_

(10) If X = 43 and Y = -22,

(a) STR\$(X + Y) = " \_\_\_\_\_ "

(b) STR\$(X) & STR\$(Y) = " \_\_\_\_\_ "

(c) VAL(STR\$(X)) + VAL(STR\$(Y)) = \_\_\_\_\_

## Answers to Chapter Twelve Exercises

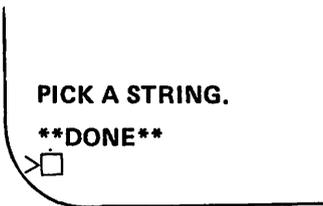
(1) 8

(2) 140 A\$ IS LONGER THAN B\$  
 160 A\$ IS SHORTER THAN B\$  
 180 A\$ IS EQUAL TO B\$

(3) 1 ( $12 - 11 = 1$ )

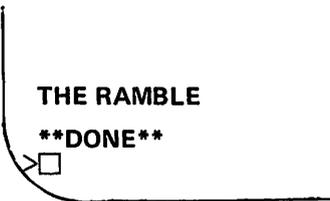
(4) RIGHT.

(5)


 A TI BASIC screen capture showing a program that has finished execution. The text on the screen reads "PICK A STRING." followed by "\*\*DONE\*\*". A cursor is visible at the end of the line.
 

PICK A STRING.  
 \*\*DONE\*\*

(6)


 A TI BASIC screen capture showing a program that has finished execution. The text on the screen reads "THE RAMBLE" followed by "\*\*DONE\*\*". A cursor is visible at the end of the line.
 

THE RAMBLE  
 \*\*DONE\*\*

(7) 65  
 66  
 67

(8) R  
 2  
 D  
 T  
 W  
 O

(9) (a) 225.16      (b) 241

(10) (a) "21"      (a string)  
 (b) "43 - 22"      (a string)  
 (c) 21      (a number)

# Chapter Thirteen

## Editing

You know how to correct a program line which has not been completely entered by:

- (1) Using the **SHIFT S** or the **SHIFT D** keys, or by
- (2) Retyping the whole line

TI BASIC also has useful tools, called **EDIT** commands, for correcting lines that have been entered. Characters may be deleted from, or inserted into, existing program lines. Let's begin by looking at how you edit a single program line.

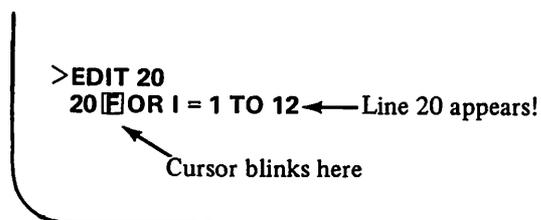
### One Line Editing

Suppose you have already entered this program.

```
10 LET A = 10
20 FOR I = 1 TO 12
30 PRINT A
40 A = A*10
50 NEXT I
```

Now, you decide to change the upper limit of the FOR-NEXT loop from 12 to 10. Instead of retyping line 20, you can correct it with the following steps.

- (1) Type: **EDIT 20**



```
>EDIT 20
20 []OR I = 1 TO 12
```

← Line 20 appears!

← Cursor blinks here

- (2) Move the cursor to the right by holding down the **SHIFT** key and pressing the **D** key.

```
EDIT 20
20 F□R I = 1 TO 12
```

↙ Cursor moves one place to the right

- (3) Continue holding down the **SHIFT** key and press the **D** key *several more times* until the cursor is blinking over the 2.

```
20 FO□ I=1 TO 12
20 FOR□I=1 TO 12
20 FOR □ = 1 TO 12
20 FOR I □ 1 TO 12
20 FOR I=□TO 12
20 FOR I=1□TO 12
20 FOR I=1 □O 12
20 FOR I=1 T□ 12
20 FOR I=1 TO□12
20 FOR I=1 TO□ 2
```

← As you continue the operation, cursor moves to the right, one step at a time

Finally, with the last key press:

```
20 FOR I=1 TO 1□ ← Cursor moves to here
```

---

*Note:* If you move too far to the right, you can back up by holding down the **SHIFT** key and pressing the **S** key until the cursor is over the 2.

---

- (4) Delete the numeral 2 by pressing the zero key.

```
20 FOR I=1 TO 10□ ← Cursor moves to next position in case
you want to add more characters
```

- (5) If this is all the editing necessary, you leave the EDIT mode by pressing the **ENTER** key.

To make sure that your edited program is correct, type **CALL CLEAR** and **LIST**. You will see the corrected result.

```

>LIST
10 LET A=10
20 FOR I=1 TO 10 ← The edited line has been corrected
30 PRINT A
40 LET A=A*10
50 NEXT I
>□

```

Let's try another one line correction. Suppose on looking over the listing, you would like to change line 40 to:

```
40 LET A=A*20
```

Once again, enter the **EDIT** mode by typing: **EDIT 40**. Line 40 appears with the cursor positioned over the **L**.

```
40 L□ET A=A*10
```

The cursor is always placed over the first character to the right of the line number when you enter the **EDIT** mode. You can then edit any part of the line from this first character to the end of the line.

If you hold down the **SHIFT** key and press the **D** key eight times, this sequence of cursor shifts will result.

```

40 L [E] T A=A*10
40 LE [T] A=A*10
40 LET [□] A=A*10
40 LET [A] =A*10
40 LET A [□] =A*10
40 LET A= [A] *10
40 LET A=A [□] *10
40 LET A=A* [□] 0

```

Since you would like to change the 1 to a 2, press the numeral two on the keyboard.

```
40 LET A=A*2[□] The cursor is now over the zero
```

Since that is the only change desired, the **ENTER** key is pressed and you leave the **EDIT** mode.

### Editing More Than One Line

While in the editing mode, you may change several program lines. When making several changes, be careful not to press the **ENTER** key after correcting the first line. That action takes you *out of* the editing mode, and you still have edit changes to make.

Enter this program after typing NEW.

```

10 CALL CLEAR
20 INPUT "FOREGROUND COLOR":F
30 INPUT "BACKGROUND COLOR:" B
40 CALL COLOR(2,F,B)
50 CALL HCHAR(12,3,42,28)
60 GO TO 55

```

Colon intentionally left out

Intentional error – GO TO a non-existent line

When the program is run, it asks for the foreground color. The number 5 is input and the **ENTER** key is pressed. The computer then detects the error in line 30.

```

FORGROUND COLOR 5
> *INCORRECT STATEMENT
 IN 30
> □

```

The procedure is similar to editing one line. You enter the **EDIT** mode.

- (1) Type: **EDIT 30** and this is what you see on the screen.

```

FOREGROUND COLOR 5
*INCORRECT STATEMENT
 IN 30
> EDIT 30
> 30: □ INPUT "BACKGROUND COLOR:" B

```

First try

Line 30 is printed so that you can look for the error

Cursor

Here is the error – no colon

- (2) Hold down the **SHIFT** key and press **D** 25 times (or until the cursor is over the **B**).

```

30 INPUT "BACKGROUND COLOR:" B

```

We are now going to insert the colon. The **B** will not be removed. When the colon is inserted the **B** is automatically shifted to the right one place to make room for the colon.

- (3) Hold down the **SHIFT** key and press **G**.
- (4) Type in the missing colon.

```
30 INPUT "BACKGROUND COLOR:" █
```

Colon is inserted and B shifted to the right to make room. Cursor is still over the B

At this point, you decide to correct the other error in line 60 while you are still in the EDIT mode. You *do not* press **ENTER**. Instead, you press **SHIFT** and the **X** key to move the line following line 30 into the edit position.

- (5) Hold down the **SHIFT** key and press the **X** key. The next line is displayed.

```
30 INPUT "BACKGROUND COLOR:":B Corrected line
40 █ ALL COLOR(2,F,B) ← Ready for editing
```

Since line 40 does not need changing, you can move on to the next line as in step 5.

- (6) While holding down the **SHIFT** key, press the **X** key again. The next line rolls into place.

```
30 INPUT "BACKGROUND COLOR:":B
40 CALL COLOR(2,F,B)
50 █ ALL HCHAR(12,3,42,28) ← This line is now ready for editing
```

Line 50 is OK also -- so you move again.

- (7) Once more, hold down the **SHIFT** key and press the **X** key. The next line rolls into place.

```
30 INPUT "BACKGROUND COLOR:":B
40 CALL COLOR(2,F,B)
50 CALL HCHAR(12,3,42,28)
60 █ O TO 55 ← This is the line you want to edit
```

- (8) Hold **SHIFT** and press **D** six times to get the cursor over the first digit in the number 55.

**60 GO TO** **5**5

- (9) Hold **SHIFT** and press **F**.

**60 GO TO** **5** ← The left 5 disappears. The right 5 shifts left one place and is now under the cursor

- (10) Hold **SHIFT** and press **F** again.

**60 GO TO** **□** ← Both 5's are gone

- (11) Type: 60 The line now shows:

**60 GO TO 60** **□**

At this point, hitting **ENTER** records all the edit changes and gets you out of the **EDIT** mode. But hold it!!

You look back at line 20 and decide that the line would look better if you added a colon after the word **COLOR**. Using the **SHIFT** key with the **E** key, you can move from one line to the preceding line. Since you are on line 60, you hold down the **SHIFT** key and press the **E** key three times.

```
60 GO TO 60
50 □ ALL HCHAR(12,3,42,28) ← Once
```

```
60 GO TO 60
50 CALL HCHAR(12,3,42,28)
40 □ ALL COLOR(2,F,B) ← Twice
```

```
60 GO TO 60
50 CALL HCHAR(12,3,42,28)
40 CALL COLOR(2,F,B)
30 □ NPUT "BACKGROUND COLOR:"":B ← Three times is the charm!
```

You are almost to line 20. But now it's your turn! You tell us how to get to line 20 and put the colon after the word COLOR.

- (1) Hold down the **SHIFT** and press the \_\_\_\_\_ key.
- (2) Hold down **SHIFT** and press \_\_\_\_\_ 23 times to get the cursor over the quote mark after the word COLOR.
- (3) Hold down **SHIFT** and press \_\_\_\_\_.
- (4) Type in the \_\_\_\_\_.
- (5) The cursor will now be over the old \_\_\_\_\_.

(Answers at the beginning of the Answers to Exercises for this chapter.)

Since you are now through EDITING, you can press **ENTER**. Remember, when you are in the EDIT mode, press **ENTER** only after you have completed all editing. This action takes you out of the EDIT mode, and you're ready to do other things. Right now you can check the program for the results of the editing changes.

Type **CALL CLEAR** and then type **LIST**. Now you see the program with all the corrections.

```
> LIST
10 CALL CLEAR
20 INPUT "FOREGROUND COLOR:":F
30 INPUT "BACKGROUND COLOR:":B
40 CALL COLOR(2,F;B)
50 CALL HCHAR(12,3,42,28)
60 GO TO 60
>□
```

### Deleting a Whole Line

Another editing command, the **SHIFTed T** key, saves both time and work. It deletes all the characters on a line except the line number. Suppose you are editing a line and you decide to start the whole line over again.

```
30 CALL COLOR(8, [E] ,B)
```

You are in the midst of editing line 30. The cursor is over the F of the color parameters

Hold down the **SHIFT** key and press **T**.

30  ← The whole line is erased, but the line number is left.  
The computer is ready for you to type in a new line 30

Try this editing command on several lines of the current program. Then press **ENTER** and **LIST** the results.

### Ignoring All Changes

Sometimes we humans have trouble making up our minds — or we change our minds in the middle of some activity. Suppose you are in the middle of editing a line and having made some changes, you decide that you didn't want to make those changes after all. TI BASIC has an editing command just for this situation. The **SHIFTed C** key signals the computer to quit editing and ignore all the changes that have been made in the line being worked on.

For example, consider a program that contains the line:

$$50 Y = (A-B)/(C-E)$$

While in the **EDIT** mode, you change this line to read:

$$50 Y = (B-A)/(E-C) \quad \boxed{\text{I}}$$

Notice that the cursor is still on line 50 and you have not left the **EDIT** mode. You realize, in reviewing the changes, that line 50 was originally correct. Rather than retype the whole line or re-edit it, you can:

Hold down the **SHIFT** key and press **C**. Line 50 will now be back in its original form.

$$50 Y = (A-B)/(C-E)$$

This key only works on the line that you are presently editing. Once you leave that line, you have to re-edit it if you want it changed.

## Automatic Line Numbering

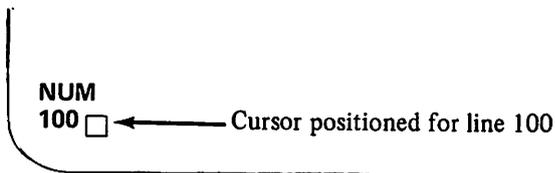
Closely associated with editing programs is the numbering of program lines. You can save some work by using the automatic line numbering feature, NUM, of your TI Home Computer. All you have to do is state your first line number and the interval you want between lines. The computer then automatically provides the number for each line. If you enter just the command NUM, the computer assumes you wish to start with line 100 and use intervals of 10.

For example:

**NUM**

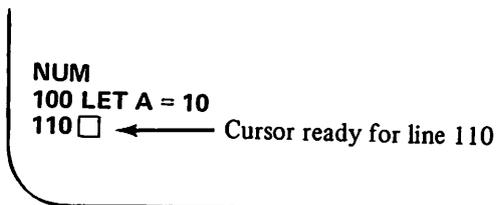
provides 100 as the first line number for your program. After line 100 has been entered, your second line is automatically numbered 110. Each succeeding line is numbered in increments of 10.

Type: NUM



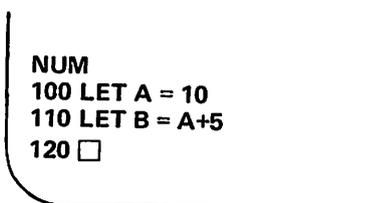
```
NUM
100 □ ← Cursor positioned for line 100
```

Type in line 100:



```
NUM
100 LET A = 10
110 □ ← Cursor ready for line 110
```

Type in line 110:



```
NUM
100 LET A = 10
110 LET B = A+5
120 □
```

Type in line 120:

```

NUM
100 LET A = 10
110 LET B = A+5
120 INPUT "C=":C
130 □

```

This process continues until your complete program is entered. If you hit **ENTER** as the first keystroke after a line number, the automatic line numbering stops.

If you wish to enter lines beginning with a line number other than 100, type **NUM** followed by the starting line number value. For example,

**NUM 10**

prompts you with a series of line numbers beginning at 10 with intervals of 10 between each line. The line numbers for this case would be 10, 20, 30 and so forth.

Typing the following command

**NUM 10,2**

would generate line numbers starting at 10 with intervals of 2 (10, 12, 14, and so forth).

### The RESequence Command

The resequence command, **RES**, works exactly like **NUM** except that it takes an existing program and completely resequences the line numbers. This operation is handy when several lines have been inserted into a program and the line numbering is becoming disorderly.

The **RES** command format is:

**RES** *starting line number, interval*

Just like **NUM**, if no starting line number and interval are specified, **RES** assumes you wish to start at 100 with intervals of 10.

Let's try **RES** on a sample program.

```

10 CALL CLEAR
20 PRINT "HELLO PEOPLE"
30 GO TO 20
RES ← Command to resequence the line numbers

```

If we now LIST the program, the screen shows:

```
>LIST
 100 CALL CLEAR
 110 PRINT "HELLO PEOPLE"
 120 GO TO 110
>□
```

What's that! Line 120 of the newly numbered program now reads

```
120 GO TO 110
```

Before using RES, it read

```
30 GO TO 20
```

Isn't that clever! The RES command is smart enough to look through a program and change the line number references that are in GO TO statements. In fact, RES will change all the line number references in GO TO, IF and any other BASIC statements automatically. RES is a powerful command.

This section completes the chapter on editing. Time to move on to the chapter summary and a set of editing exercises.

## Summary of Edit Commands

The **SHIFT** key is used with all of the **EDIT** keys.



Move to line preceding the present line.



Move to line following the present line.



Move left one position.



Move right one position.



Delete the character under the cursor. All characters to the right of the cursor will move one place to the left.



Insert a number of characters. As characters are inserted, all those to the right of the cursor will move one place to the right for each new character inserted. The cursor and the character under it also move right one place.



Deletes all the characters on the current line except for the line number.



Break — quit editing. The changes in the line being edited are ignored.

**ENTER** Quit editing and accept all changes as permanent.

You also covered the line numbering commands **NUM** and **RES**. **NUM** generates line numbers and line number intervals for you. This command reduces your typing efforts. **RES** will resequence an entire program's line numbering, including any references to line numbers in **GO TO** and **IF** statements. Both commands have the same format:

**NUM** *starting line number, interval*

**RES** *starting line number, interval*

If no starting line number and interval is specified, both commands start with line number 100 and use intervals of 10. If only the starting line number is given, the interval is assumed to be 10.

## Chapter Exercises

(1) When editing a line, which key (with **SHIFT** held down)

(a) Moves the cursor to the right? \_\_\_\_\_

(b) Moves the cursor to the left? \_\_\_\_\_

(2) If the **SHIFT** key is held down and the **D** key is pressed five times and the **S** key twice, where would the cursor be in line 50?

\_\_\_\_\_

**50 PRINT "ANI  AL CRACKERS"**

(3) You enter this program.

```
100 CALL CLEAR
110 LET A = 1
120 PRINT "A = ";A
130 A = A+1
140 GO TO 120
```

You then type **CALL CLEAR**, followed by **EDIT 110**. Then press **X** twice while holding down the **SHIFT** key. Show what is now on the screen.



(4) Complete the blanks below. Tell how to edit line 130 in exercise 3 so that the program prints only *odd* integers (1,3,4, etc.).

**SHIFT**  \_\_\_\_\_ times

Then type:

**ENTER**

(5) You are editing line 60 below.

**60 PRINT "ICE C  EAM CONES"**

You hold down **SHIFT** and press **T**. Describe the results: \_\_\_\_\_

\_\_\_\_\_

- (6) You wish to edit line 110 of the program below. You are already in the EDIT mode with the cursor on line 120 as shown.

```
100 CALL CLEAR
110 PRINT "ICE CRAEM"
120 PRINT "IS COOL"
```

```
120 PRINT "IS OOL"
```

What is the quickest way to get to line 110 for editing? \_\_\_\_\_

---

- (7) You are now at line 110 of the exercise 6 program and want to spell ICE CREAM correctly. Fill in the blanks below to make the correction.

SHIFT  \_\_\_\_\_ times Type:   and press:

- (8) Now that the program has been corrected in exercise 7, how do you get out of EDIT mode?
- 

- (9) What line numbers would be automatically provided for a five-line program by:

NUM 200,5 \_\_\_\_\_, \_\_\_\_\_, \_\_\_\_\_, \_\_\_\_\_, and \_\_\_\_\_

**Answers to Questions on Page 259**

(1) E (2) D (3) E (4) : (5) “

**Answers to Exercises**

(1) (a) D (b) S

(2) Over the blank between the words ANIMAL and CRACKERS.

(3) 110 LET A = 1  
120 PRINT "A = ";A  
130 A = A+1

(4) D 4 times  
Then type: 2 ENTER

(5) Line 60 is erased except for the line number. Just the cursor appears to the right of the line number.

60□

(6) Hold down SHIFT and press E.

(7) D 13 times  
E  
A  
ENTER

(8) Press ENTER

(9) 200, 205, 210, 215 and 220

# Chapter Fourteen

## Subroutines and Your Personal Library

You are now in the last chapter of this book. You have covered a lot of information on your TI Home Computer and its BASIC language. You have tried many small programs and sets of statements on the machine. You have learned how to do some fundamental programming activities.

What is next? Well, it's time for you to start thinking of the uses and recreations that you might create for yourself. What might those be? They could be anything – anything you would like to have your Home Computer do for you. The selection ranges from games and simulations to programs that would be useful in helping you manage and run your home.

A logical place to start building programs for yourself is in those areas that hold an interest for you. If you are interested in games, then you might begin there. If your interests are in using the computer for record keeping, then begin in that area. But wherever you choose to begin, you will find that there are certain routines and procedures that you will need over and over again. That set of routines is what this chapter is about.

In this chapter, we will develop a set of subroutines that you may find useful to have in your personal software library. Some procedures you have seen before in the early chapters of this book. Those routines will be repeated here, but with changes.

We will, in this chapter, adopt some conventions for specifying subroutines so that the variables in the main program and those in the subroutine do not conflict.

In addition, we'll include some new procedures that have not been discussed in the book. Taken together, these old and new routines will form the basis for your personal software library.

Let's start with a familiar subroutine, the *delay* operation.

## The Delay Subroutine

Earlier in this book, you learned to use a FOR-NEXT loop to delay actions on the screen. Let's now formalize this fundamental operation as a subroutine that could be used in any program. We begin with the delay subroutine because you are familiar with how it works, and it provides an uncomplicated example of how we wish to set up variable name conventions.

For variables that are used only within a subroutine, we will adopt the convention that they are named as follows:

Numeric variables:        **SUBn**  
 String variables:        **SUBn\$**

where the n in the name is a number. Thus, SUB1, SUB2, and so forth are subroutine numeric variable names. SUB1\$, SUB2\$, and so on are subroutine string variable names.

For variables that are used in both the main program and the subroutine, we will attempt to use names that are meaningful to whatever action is being performed. For example, in the delay subroutine and main program, we might have cause to use a variable called DELAY.

Why are we taking such pains to set up formal variable name conventions? We want to minimize the chances for variable name conflicts between the main program and the subroutines. Such conflicts can cause subtle and difficult-to-find errors in your programs.

The delay routine that you have been using looks something like this:

```
1000 REM**DELAY SUBROUTINE**
1010 FOR SUB1 = 1 TO 200 ← Delay count
1020 NEXT SUB1 ← We are using our variable naming convention
1030 RETURN
```

For this routine to be as general as possible, the delay count needs to be specified as a parameter that is passed into the subroutine. In that way, the procedure can be used from within many parts of the main program and other subroutines. If we change the routine, making the delay count a parameter, the subroutine becomes:

```
1000 REM**GENERAL DELAY SUBROUTINE**
1010 FOR SUB1 = 1 TO DELAY ← Delay count is now a parameter
1020 NEXT SUB1
1030 RETURN
```

The calling sequence for this subroutine might be:

```
100 DELAY = 200 ← Set the delay count
110 GOSUB 1000 ← Call the delay routine
```

Do you see how we are constructing our library of routines? Again, the key elements of what we are doing are:

- (1) The use of SUB as the first three letters of the variable names that are used exclusively within the subroutines. The three letters are followed by a number, SUB1, SUB2, and so forth, to distinguish between various individual variable names.
- (2) In the main program, we use a name that is as meaningful as possible, for parameters that are being passed to and from the subroutine. For the first routine discussed, a delay operation, we used the parameter named DELAY.

Are there any variations of this subroutine that may be useful to have in your library? What about a version that performs a delay and then clears the screen?

### Delay and Clear Subroutine

In several programs that we have constructed, we had a need to implement a delay to hold some piece of information temporarily on the screen. Then we cleared the screen. You might find it handy to have this combined operation represented as a single subroutine.

```

1100 REM**DELAY AND CLEAR SUBROUTINE**
1110 FOR SUB1 = 1 TO DELAY
1120 NEXT SUB1
1130 CALL CLEAR ←————— Add the screen clear command
1140 RETURN

```

This routine can be used as part of programs that “flash” characters or information on the screen, and that do graphical “animation” of characters.

The calling sequence for this routine looks just like that for the regular delay operation:

```

100 DELAY = 300
110 GOSUB 1100 ←————— Call delay and clear routine

```

Let’s construct an example using the last two subroutines. Below is the calling sequence that “flashes” the character (\*) near the middle of the screen.

```

100 CALL CLEAR
110 CALL VCHAR(12,16,42) ←————— Put asterisk on screen
120 DELAY = 300 ←————— Set delay parameter
Set parameter to new value → 130 GOSUB 1100 ←————— Hold the character on the screen for a count of 300, and then clear the screen
140 DELAY = 200
150 GOSUB 1000 ←————— Leave the screen blank for a count of 200
160 GO TO 110 ←————— Return to line 110, and redisplay asterisk

```

Now that we have set the groundwork concerning naming conventions, let's move on to more interesting subroutines. We have much to choose from given the color, sound, and graphical capabilities of the TI Home Computer. Also, the string handling functions provide us with other opportunities to construct library routines for you. We begin with some graphical routines.

### Putting a Border Around the Screen

A useful routine, which can be called upon several times according to what is occurring in the main program, is one that draws or redraws a border at the edge of the screen. The use of a border to highlight what is on the screen is mainly cosmetic, but it does lend a nice visual emphasis to whatever you place within its boundaries.

The subroutine that draws this border is given below:

```

1200 REM**SCREEN BORDER ROUTINE**
1210 CALL CLEAR
1220 CALL CHAR(96,"FFFFFFFFFFFFFF") ← Redefine character 96 to be ■
1230 CALL HCHAR(1,3,96,26) }
1240 CALL HCHAR(24,3,96,26) } ← Draw top and bottom lines
1250 CALL VCHAR(2,3,96,22) }
1260 CALL VCHAR(2,28,96,22) } ← Draw sides of border
1270 RETURN

```

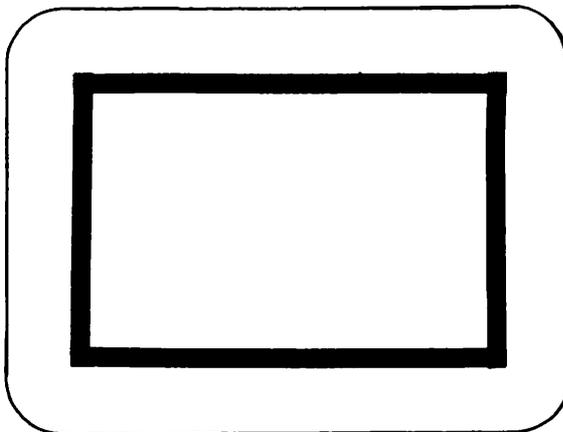
The calling sequence for this routine is quite simple:

```
100 GOSUB 1200
```

You might enter this routine into your computer and give it a try. Add the following line to hold the border on the screen so that you can view what the drawing looks like:

```
110 GO TO 110
```

The screen will show:



The use of CALL CHAR in the last routine suggests a couple of other subroutines. Let's examine one such procedure.

### Redefining a Block of Characters

There may be occasions when you will need to redefine a block of graphics characters with the CALL CHAR routine. If the string values for the characters are read from DATA statements into a string array, and the number of characters to be redefined is known, then a single subroutine can handle the redefinition.

We will assume that the string values are in an array CHARCODE\$, and that NUMCHARS contains the value for the number of characters to be redefined. The variable, STARTCHAR, tells the routine where to begin the block of new characters. The calling sequence might be:

```
100 NUMCHARS = 10 ← Number of characters being redefined
110 STARTCHAR = 96 ← Begin with character code 96
120 GOSUB 1300 ← Redefine block of characters
```

The subroutine is given below:

```
1300 REM**BLOCK OF CHARACTERS – REDEFINITION**
1310 FOR SUB1 = 0 TO NUMCHARS – 1
1320 CALL CHAR(STARTCHAR + SUB1, CHARCODE$(SUB1+1))
1330 NEXT SUB1
1340 RETURN
```

Character code
String of data representing  
the new character

For the example, a block of ten characters is redefined by the subroutine. The new characters are those with character codes 96 through 105. The string values that are used to describe the new codes are taken from the array CHARCODE\$. The single loop within the subroutine accomplishes the entire redefinition.

You might like to experiment with this subroutine. Set up a DATA statement block that contains the string values for the new characters. Read the data into the CHARCODE\$ array. Call the subroutine to redefine the characters. Then display the characters using the VCHAR and HCHAR routines.

### Using VCHAR/HCHAR to Reward a Successful Guess

In learning programs or games that have a correct answer, the use of a subroutine to perform a graphical “fireworks” display when a correct guess is made adds interest to the program. The subroutine shown below clears the display, and randomly places 100 asterisks all about the screen. A message is printed at the bottom of the screen that indicates the number of guesses that were made. The program then uses the

CALL KEY routine to hold the information on the screen while waiting for the user to signal that the program can continue. The parameter GUESS is assumed to have been set in the main program and to contain the number of guesses the player has made. The calling sequence is therefore quite simple:

**100 GOSUB 1400**

The subroutine follows:

```

1400 REM**ROUTINE TO REWARD A GUESS**
1410 CALL CLEAR
1420 FOR SUB1 = 1 TO 100
1430 SUB2 = INT(24*RDN)+1 ← Pick a row
1440 SUB3 = INT(32*RND)+1 ← Pick a column
1450 CALL VCHAR(SUB2,SUB3,42) ← Put an asterisk at the random location
1460 NEXT SUB1
Reward { 1470 PRINT "CONGRATULATIONS!!"
message } 1480 PRINT "YOU GOT IT IN ";GUESS;" GUESSES!!"
1490 PRINT "PRESS ANY KEY TO CONTINUE:";
1500 CALL KEY(0,SUB1,SUB2) } ← Wait for key to be pressed. SUB1 will contain
1510 IF SUB2 = 0 THEN 1500 } ← character code of key. SUB2 is status indicator.
1520 RETURN } ← If no key is pressed it is zero

```

The routine just shown should work with any form of guessing game. You might consider a couple of other routines that use the sound and color capabilities of the Home Computer.

### A Sound and Color Subroutine

For example, here is a sound and color subroutine that can be called upon as a reward for a correct answer, as a signal that something in the main program is about to happen, or as a warning signal to the user that the last move they made put them in danger. The routine uses the CALL SCREEN operation to "flash" the background colors on the monitor. As the colors flash, a tone is produced that oscillates up and down, over a range of frequencies. For the routine shown, no messages are printed. You can add whatever messages you might need to the routine, according to how it is being used by the main program.

The calling sequence is:

**100 GOSUB 1600**

The subroutine that is based on color and sound is given below:

```
1600REM
```

```
1600 REM**COLOR AND SOUND WARNING/REWARD ROUTINE**
```

```
1610 CALL CLEAR
```

```
1620 FOR SUB1 = 1 TO 5
```

```
1630 SUB2 = 100 ← Duration parameter
```

```
1640 FOR SUB3 = 220 TO 440 STEP 40 ← Notes going up
```

```
1650 CALL SCREEN(INT(16*RND)+1) ← Random color change
```

```
1660 CALL SOUND(SUB2,SUB3,2) ← Play note
```

```
1670 SUB2 = SUB2 + 50 ← Extend duration
```

```
1680 NEXT SUB3
```

```
1690 FOR SUB3 = 440 TO 220 STEP -40 ← Notes going down
```

```
1700 CALL SCREEN(INT(16*RND)+1) ← Random color change
```

```
1720 CALL SOUND(SUB2,SUB3,2) ← Play note
```

```
1730 SUB2 = SUB2 - 50 ← Decrease duration
```

```
1740 NEXT SUB3
```

```
1750 NEXT SUB1
```

```
1760 RETURN
```

The loop (lines 1620 to 1750) that controls how many times the sets of notes are played and color changes are made.

There are at least two variations of this last routine that you may want to construct for yourself:

- (1) Color only — a subroutine that does nothing but flash the background colors randomly. This procedure could be used, for example, as a signal to the user that an incorrect entry was made.
- (2) Sound only — a routine that “echoes” a sound when keys are pressed on the keyboard or that initiates a sequence of random tones when an incorrect key is pressed.

Routines like these enliven your programming efforts. They are also handy to use in programs that are designed for small children who cannot read yet. The children quickly learn to respond to visual and aural signals from the computer.

You may also want to experiment with the last subroutine dealing with both color and sound. How would you change the sound portions of the program so that the notes *ululate* (oscillate rapidly up and down on each successive note to produce a wailing or siren-like sound)? Is it possible to make a sound that resembles a space drive engine or a laser gun? You could build yourself a library of *sound* routines that you could use to enhance your programs. How does that *sound* to you?

### Removing Spaces in a String

In many applications involving strings of data, you may want to eliminate all the spaces in a string before you use it. The next routine does just that — removes the spaces from a string of characters. The string variable from the main program is assumed to be named `STRING$`.

The calling sequence might appear as follows:

```
100 STRING$ = "THIS LINE WILL HAVE NO SPACES"
110 GOSUB 2000
120 PRINT STRING$
```

The PRINT statement is included so that you can verify that the spaces have been removed. You may not want to include line 120 in your regular programs that use this routine.

The subroutine that is used to squeeze the spaces from the line of data looks at each character in STRING\$. If it is a space, the routine skips over the character. If the character is not a space, it is concatenated (attached) to the other nonblank characters that have been found. The space-less resultant string is placed back into STRING\$.

*Note:* This operation destroys the original string.

The string squeeze routine is given below:

```
2000 REM**SPACE REMOVAL SUBROUTINE**
2010 SUB1 = LEN(STRING$) ← Get length of STRING$
2020 IF SUB1 = 0 THEN 2110 ← Check for an empty string
2030 SUB2$ = "" ← Set to empty string
Loop over all characters → { 2040 FOR SUB3 = 1 TO SUB1
 2050 SUB4$ = SEG$(STRING$,SUB3,1) ← Extract a character
 2060 IF SUB4$ = " " THEN 2080 ← Check for space
 2070 SUB2$ = SUB2$ & SUB4$ ← Concatenate nonblank character
 2080 NEXT SUB3
 2090 STRING$ = SUB2$ ← Put squeezed string back into STRING$
 2100 SUB2$ = "" ← Set back to empty string
2110 RETURN
```

If you enter and RUN this routine, including the calling sequence, the following line of data should appear on the screen:

```
THISLINEWILLHAVENOSPACES
```

There are several modifications of this subroutine that might be of use to you. You can have the routine drop any character from a line of data by altering line 2060. In fact, if you made line 2060 a function of a parameter that is passed from the main program, then the routine would search out and drop any characters you wished. Try the modification yourself and see what occurs.

Also, instead of dropping characters, you could make the routine add or change characters. That routine might be handy at times. If you had a function that would change any character to any other character or set of characters, you would have one part of a *text editing system*.

But how about another routine that removes all characters that are not part of the uppercase alphabet?

### Removing Sets of Characters from a String

This subroutine can be used to “clean up” a string of “dirty” data caused by poor typing or errors from a cassette tape device. The procedure is similar to that of removing blanks. The difference between the two routines is the use of the ASC function in this program to exclude all non-uppercase letters.

The calling sequence is:

```
100 STRING$ = "DR%$OPTHE!@1FU--NNYST*&UUF"
110 GOSUB 2200
120 PRINT STRING$
```

Note once again, the variable STRING\$ is altered by the subroutine. The original content of STRING\$ is lost when the program is executed.

Here is the subroutine for removing all non-uppercase letters:

```
2200 REM**REMOVE ALL NON-UPPERCASE LETTERS ROUTINE**
2210 SUB1 = LEN(STRING$)
2220 IF SUB1 = 0 THEN 2320
2230 SUB2$ = ""
2240 FOR SUB3 = 1 TO SUB1
2250 SUB4$ = SEG$(STRING$,SUB3,1)
2260 IF ASC(SUB4$)<65 THEN 2290 } Check for non-upper
2270 IF ASC(SUB4$)>90 THEN 2290 } case characters
2280 SUB2$ = SUB2$ & SUB4$
2290 NEXT SUB3
2300 STRING$ = SUB2$
2310 SUB2$ = ""
2320 RETURN
```

Except for lines 2260 and 2270, this routine is similar to Space Removal subroutine.

If you enter and RUN this program, the display will show:

**DROPTHEFUNNYSTUFF**

That's right! The subroutine has *dropped the funny stuff* from the original line of data. What else could you do with this routine? Yes, as in the Space Removal Subroutine, you could change this one to add or alter characters. You could also change lines 2260 and 2270 so that only numbers would be kept – or numbers and some letters – or even have it exclude letters but keep numbers and special characters.

But, enough on strings! Let's move on to other subroutines.

## Rolling Dice

If there are any fantasy gaming people reading this book, here is a subroutine for you. The routine simulates the rolling of a set of dice. You can roll up to ten dice at a time. Each die can have any number of faces. The routine returns to your main program with the values for the individual die, and a total for all dice rolled.

The calling sequence sets up NUMDICE and FACES. NUMDICE is the number of dice to be rolled. FACES tells the routine how many faces there are on each die. The return parameters are DICE, an array of values for the dice rolled, and TOTAL, the sum of all the rolls. The calling sequence for a roll of five eight-sided dice is:

```
100 NUMDICE = 5 ← Set number of dice
110 FACES = 8 ← Specify faces on each die
120 GOSUB 2400
```

The RND function is used to select at random the values for each roll. The subroutine looks like this:

```
2400 REM**DICE ROLLING ROUTINE**
2410 SUB1 = 0 ← Total of all rolls goes in SUB1
2420 FOR SUB2 = 1 TO NUMDICE
2430 SUB3 = INT(FACES*RND)+1 ← Roll one die
2440 SUB1 = SUB1 + SUB3 ← Accumulate total
2450 DICE(SUB2) = SUB3 ← Put roll in array
2460 NEXT SUB2
2470 TOTAL = SUB1 ← Put total in main program variable
2480 RETURN
```

What would the calling sequence look like for one roll of a 100 sided die?

```
100 NUMDICE = 1 ← One die
110 FACES = 100 ← 100 sided
120 GOSUB 2400
```

This subroutine will give the same values for a sequence of rolls unless the RANDOMIZE statement is used. You can put RANDOMIZE in the calling sequence if you want each set of rolls to be completely different.

If some of you are wondering what *fantasy gaming* is, just ask the next child you see to tell you. Fantasy gaming and fantasy role playing (FRP) games are quite popular throughout the country. There are thousands of people (adults and kids) who regularly play *Dungeons & Dragons* (D&D). There are clubs, magazines (*fanzines*), special props, and many other activities and specialized materials for the game. Check with your local hobby store for details.

Let's end our library tour of subroutines with some music.

### A Music Maker Subroutine

The music features of your Home Computer are quite unique. Music adds to the entertainment value of your machine. You can also use sound and music to make programs more interesting.

It is possible to write one subroutine that will play nearly any of your "compositions." Using arrays to hold the data needed by the SOUND routine, you can use a single subroutine as the heart of your music making.

Recall the SOUND routine? It can play from one to three notes. It looks like this:

```
CALL SOUND(duration,note 1, loudness 1, note 2, loudness 2, note 3, loudness 3)
```

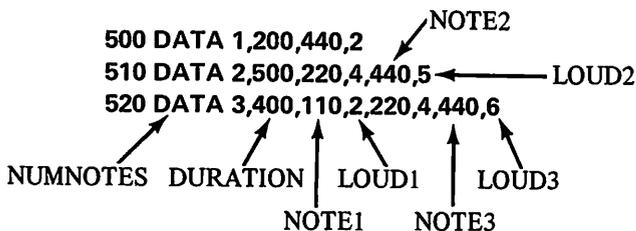
If we used several arrays to hold a set of durations, notes, and loudness values, we could play all those notes using just three calls on the SOUND routine. The data for the "melody" could be stored in DATA statements and READ into the arrays. "Tunes" and "songs" could then be selected, read into the arrays, and played by the single subroutine. *Sounds nice, doesn't it?* Let's see how we could do it.

The reason it would take three calls on SOUND is that SOUND can play either one, two, or three notes at a time. Thus, part of our data would have to contain the information on how many notes were to be played simultaneously.

For arrays, we would therefore need:

- NUMNOTES An array of values that tell how many notes are to be played for each call on SOUND.
- DURATION An array of duration values for each call on SOUND.
- NOTE1 An array of frequency values for the first note to be played.
- LOUD1 A loudness value for the notes in NOTE1.
- NOTE2 A second set of notes.
- LOUD2 Loudness values for NOTE 2 notes.
- NOTE3 A third array of notes.
- LOUD3 Loudness values for NOTE3 notes.

We assume that an appropriate set of data statements has been used to fill the arrays. A typical set of data might look like the following:



Only one other parameter is needed for the subroutine – the number of times CALL SOUND is to be activated. We will use the variable NOTECOUNT for this parameter. It will contain the number of rows of DATA statements that we used to set up the arrays. In this case, NOTECOUNT will be set to three.

The calling sequence is:

```
100 NOTECOUNT = 3
110 GOSUB 2600
```

If needed, the main program is assumed to contain a DIM statement for declaring all the arrays that are used in this example.

Let's look at the subroutine:

```
2600 REM**MUSIC MAKER SUBROUTINE**
2610 FOR SUB1 = 1 TO NOTECOUNT
One note { 2620 IF NUMNOTES(SUB1)>1 THEN 2650
 { 2630 CALL SOUND(DURATION(SUB1),NOTE1(SUB1),LOUD1(SUB1))
 { 2640 GO TO 2700
Two notes { 2650 IF NUMNOTES(SUB1)>2 THEN 2680
 { 2660 CALL SOUND(DURATION(SUB1),NOTE1(SUB1),LOUD1(SUB1),
 { NOTE2(SUB1),LOUD2(SUB1))
 { 2670 GO TO 2700
Three notes { 2680 IF NUMNOTES(SUB1)<>3 THEN 2700
 { 2690 CALL SOUND(DURATION(SUB1),NOTE1(SUB1),LOUD1(SUB1),
 { NOTE2(SUB1),LOUD2(SUB1),NOTE3(SUB1),LOUD3(SUB1))
 { 2700 NEXT SUB1
 { 2710 RETURN
```

Given a set of data in the arrays, this subroutine will play them all. The routine will work for any set of notes and sound parameters. Why don't you compose a tune, enter it in DATA statements, READ the data into the arrays, and have the Music Maker Subroutine play for you? Have fun!!



**What's Next in Adventureland?**

Many pages ago, you entered the gateway to this adventureland with your Home Computer. You have traveled widely in this land, visited places, and accumulated many experiences about computing and computers.

Now, it's the beginning of the real adventures. We have enjoyed our brief stint serving as your guide through the BASIC language, the TI computer's capabilities, and some fundamental programming hints. But it's your turn. You have been shown enough so that you can begin to experiment with the computer on your own. It's time for you to create adventures for yourself.

Oh, yes! We forgot to mention something when we entered the gateway back at the beginning of the book. Once you have entered the adventureland of Home Computing, There is no simple way out. If you have worked your way through this book, picking up the techniques for using your Home Computer as you did so, you will probably never want to return to the starting point again. Most likely, you will want to continue to explore and investigate more of this land. Who knows – you may even discover some parts of this adventureland that no one has ever seen.

Enjoy your explorations. Perhaps we will bump into each other again as you pursue your adventures.

## Chapter Summary

In this chapter, the last chapter in the book, you have started the process of building your personal software library. You have looked at a variety of subroutines that might possibly be included in the library of programs. You have looked at the following procedures:

- Delay Subroutine
- Delay and Clear Screen Subroutine
- Putting a Border on the Screen
- Redefining a Block of Characters
- Rewarding a Successful Guess with Asterisks
- Rewarding a Guess with Sound and Color
- Space Removal Subroutine
- Removing Sets of Characters from a String
- Dice Roll Subroutine
- Music Maker Subroutine

In addition, you were given hints on how to change the current set of subroutines into several other procedures that you might find useful.

You probably feel as if you have covered quite a lot of material in just this chapter alone. You are right! You have. This chapter brought together many of the BASIC language features that were discussed throughout the rest of the book. So you had to work hard to make it through these last few pages.

The authors are aware of this fact and have prepared a surprise for you. *There is no set of exercises at the end of this chapter.* You have graduated. Good luck and goodbye!

# Appendix A

## Musical Notes and Frequencies

The tones produced by the Texas Instruments Home Computer are generated by the CALL SOUND statement. (See Chapter 1 for an explanation of the CALL SOUND statement).

The *frequency* designated in the CALL SOUND statement determines the tone that is produced. The acceptable value range for frequencies is from 110 to 44,733 Hertz (cycles per second). Noninteger entries within this range are acceptable as inputs in the CALL SOUND statement, but they are rounded to nearest integers by the computer before execution.

The following table gives frequency values (rounded to integers) for four octaves in the tempered scale (one half-step between notes). While these values do not, of course, represent the entire range of tones — or even of musical tones — they can give you a basis for musical programs. (See Appendix D for a frequency-generating program.)

| Frequency | Note               | Frequency | Note               |
|-----------|--------------------|-----------|--------------------|
| 110       | A                  | 440       | A (above middle C) |
| 117       | A#.B               | 466       | A#.B               |
| 123       | B                  | 494       | B                  |
| 131       | C (low C)          | 523       | C (high C)         |
| 139       | C#.D               | 554       | C#.D               |
| 147       | D                  | 587       | D                  |
| 156       | D#.E               | 622       | D#.E               |
| 165       | E                  | 659       | E                  |
| 175       | F                  | 698       | F                  |
| 185       | F#.G               | 740       | F#.G               |
| 196       | G                  | 784       | G                  |
| 208       | G#.B               | 831       | G#.A               |
| 220       | A (below middle C) | 880       | A (above high C)   |
| 220       | A (below middle C) | 880       | A (above high C)   |
| 233       | A#.B               | 932       | A#.B               |
| 247       | B                  | 988       | B                  |
| 262       | C (middle C)       | 1047      | C                  |
| 277       | C#.D               | 1109      | C#.D               |
| 294       | D                  | 1175      | D                  |
| 311       | D#.E               | 1245      | D#.E               |
| 330       | E                  | 1319      | E                  |
| 349       | F                  | 1397      | F                  |
| 370       | F#.G               | 1480      | F#.G               |
| 392       | G                  | 1568      | G                  |
| 415       | G#.A               | 1661      | G#.A               |
| 440       | A (above middle C) | 1760      | A                  |

# Appendix B

## Character Codes

All characters that print on the screen (letters, numbers, and symbols) are identified by numeric *character codes*. The standard characters are represented by character codes 32 through 95. These 64 codes are grouped into eight *character sets* for color graphic purposes.

### Standard Character Codes

| Set #1 |           | Set #2 |           | Set #3 |           | Set #4 |           |
|--------|-----------|--------|-----------|--------|-----------|--------|-----------|
| Code#  | Character | Code#  | Character | Code#  | Character | Code#  | Character |
| 32     | (space)   | 40     | (         | 48     | 0         | 56     | 8         |
| 33     | !         | 41     | )         | 49     | 1         | 57     | 9         |
| 34     | "         | 42     | *         | 50     | 2         | 58     | :         |
| 35     | #         | 43     | +         | 51     | 3         | 59     | :         |
| 36     | \$        | 44     | .         | 52     | 4         | 60     | <         |
| 37     | %         | 45     | -         | 53     | 5         | 61     | =         |
| 38     | &         | 46     | '         | 54     | 6         | 62     | >         |
| 39     | ,         | 47     | /         | 55     | 7         | 63     | ?         |
| Set #5 |           | Set #6 |           | Set #7 |           | Set #8 |           |
| Code#  | Character | Code#  | Character | Code#  | Character | Code#  | Character |
| 64     | @         | 72     | H         | 80     | P         | 88     | X         |
| 65     | A         | 73     | I         | 81     | Q         | 89     | Y         |
| 66     | B         | 74     | J         | 82     | R         | 90     | Z         |
| 67     | C         | 75     | K         | 83     | S         | 91     | [         |
| 68     | D         | 76     | L         | 84     | T         | 92     | /         |
| 69     | E         | 77     | M         | 85     | U         | 93     | ]         |
| 70     | F         | 78     | N         | 86     | V         | 94     | ^         |
| 71     | G         | 79     | O         | 87     | W         | 95     | _         |

There are 64 additional character codes (96-159) available for use in defining special characters for graphics programs. (See Chapter 5 for a discussion of character definition.) Again, these codes are grouped into eight sets for color graphics.

### Special Character Codes

| Set #9 | Set #10 | Set #11 | Set #12 | Set #13 | Set #14 | Set #15 | Set #16 |
|--------|---------|---------|---------|---------|---------|---------|---------|
| 96     | 104     | 112     | 120     | 128     | 136     | 144     | 152     |
| 97     | 105     | 113     | 121     | 129     | 137     | 145     | 153     |
| 98     | 106     | 114     | 122     | 130     | 138     | 146     | 154     |
| 99     | 107     | 115     | 123     | 131     | 139     | 147     | 155     |
| 100    | 108     | 116     | 124     | 132     | 140     | 148     | 156     |
| 101    | 109     | 117     | 125     | 133     | 141     | 149     | 157     |
| 102    | 110     | 118     | 126     | 134     | 142     | 150     | 158     |
| 103    | 111     | 119     | 127     | 135     | 143     | 151     | 159     |

# Appendix C

## Color Codes

Sixteen colors are available for color graphics programs in TI BASIC. These colors are designated by numeric codes in the CALL COLOR and CALL SCREEN statements. (See Chapter 2 for a discussion of CALL COLOR and Chapter 5 for an explanation of CALL SCREEN.)

## 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 D

## Mathematical Operations

If your computer is to be a useful tool, you'll need to know about some of its computational powers. This appendix first discusses the ways your computer handles and displays numbers and then shows you how to perform exponentiation (powers and roots of numbers). Next is a section on the order in which mathematical operations are performed. Finally, certain other mathematical functions are listed for you. You'll find that your computer can eliminate much of the drudgery of computation leaving you with more time to explore the theory and fun of mathematics.

## Decimal Notation

The Texas Instruments Home Computer accepts and displays numbers, within certain limits, in the traditional decimal form.

In Chapter 3, we mentioned briefly that numbers are displayed with a *leading space* and a *trailing space*. The *leading space* is reserved for the *sign* (positive or negative) of the number. If the number is positive, this space will be blank. If the number is negative, this space will show a minus sign. Here's an example of both situations:

```
>PRINT 1
 1
>PRINT -1
-1
>□
```

The *trailing space* is there to make sure that two numbers on the same line of the screen will always have at least one space between them, even if you use a semicolon as a PRINT separator. (The semicolon instructs the computer to leave no spaces between PRINT items.) Try this Immediate Mode example to see the effect of the trailing space:

```
>PRINT 1;-1
 1 -1
>□
```

Without this trailing space the two numbers would appear like this:

```
1-1
```

The screen shows a maximum of ten digits for any number. If an integer (whole number) consists of ten digits or less, the computer shows the number *without* a decimal point to the right:

```
>PRINT 1;12345;1234567890
1 12345 1234567890
>□
```

If the number is a decimal fraction with ten digits or less, the computer automatically places the decimal point in the correct position:

```
>PRINT 1/8;7.525/5;159.1395/5
.125 1.505 31.8279
>□
```

Notice the first example above,  $1/8 = .125$ . If a number is less than +1 and greater than -1, so that digit to the left of the decimal point would be zero, the zero is not displayed.

Most of the time, the numbers you see and work with will be shown in this normal display format. But what about numbers that consist of *more* than ten digits, such as

```
723,895,274,
0.00000000014896
```

The computer can also handle numbers like these, but it must use a special display format to do so.

### Floating Point Form or Scientific Notation

To display numbers with more than ten digits, your computer uses a special kind of notation. You'll see several names in computer books referring to this type of notation: two of the more common names are *floating point form* and *scientific notation*. Here we'll refer to the special display format as scientific notation.

Before we discuss scientific notation, let's try a program to see how whole numbers (integers) look in this display format. Enter these lines:

```
NEW
10 LET A=10
20 FOR I=1 TO 12
30 PRINT A
40 LET A=A*10
50 NEXT I
60 END
```

Now clear the screen and run the program. You'll see these results:

```

10
100
1000
10000
100000
1000000
10000000
100000000
1000000000
1.E+10 ← Scientific notation starts here
1.E+11
1.E+12

```

As soon as the value of A becomes an integer with more than ten digits, the computer switches over to the special display format. Here's what this format represents:

```

1.E+10 means 1x1010 or 10,000,000,000
1.E+11 means 1x1111 or 100,000,000,000
1.E+12 means 1x1012 or 1,000,000,000,000

```

Numbers that are printed in scientific notation will always have this form:

base number E exponent

The base number (*mantissa*) is always displayed with one digit (1 through 9) to the left of the decimal point. There can be a maximum of six digits in the mantissa (one to the left of the decimal point, up to five to the right of the decimal). "E" stands for "x (times) 10 raised to some power," and the *exponent* (power) is always displayed with a plus or minus sign (+ or -) followed by a one- or two-digit number (1 through 99).

The two asterisks indicate that the number is within the valid computing range of the computer, but the exponent is too large to be displayed in the allotted space. (For a discussion of the computational ranges, see the BASIC Reference section of the *User's Reference Guide*.)

Here are several examples of integers that are displayed by the computer in scientific notation:

```

>PRINT 1234512345123
 1.23451E+12

>PRINT 45678900000000
 4.56789E+13

>PRINT 98765432100
 9.87654E+10

>□

```

Notice that the *sign* of the exponent tells us how to convert scientific notation back into standard decimal form. If the sign is a +, we move the decimal point to the right. If the sign is a -, we move the decimal point to the left. The *exponent* tells us how many places to move the decimal point:

1.11111E+10 means 1111100000

We have moved the decimal ten places to the right:

1111100000.

Integers with more than ten digits, then, are always displayed in scientific notation. Now let's see how the computer handles noninteger numbers (numbers with fractional parts). Consider the number 0.000000000000123. Since it will not fit into the ten-digit display, the computer shows it in scientific notation. Try this:

```

>PRINT 0.000000000000123
1.23E-13 ← Tells how many places to move
 the decimal to the left
>□

```

The following program generates some very small noninteger numbers:

```

NEW
10 LET A=10
20 FOR I=1 TO 14
30 PRINT A
40 LET A=A/10
50 NEXT I
60 END

```

Clear the screen and run the program. The results are:

```

10
1
.1
.01
.001
.0001
.00001
.000001
.0000001
.00000001
.000000001
.0000000001
1.E-11
1.E-12

```

This program and the previous examples we've seen might lead us to think that nonintegers with more than ten digits are always displayed in scientific notation, just as integers are. This is not always true, however. *Noninteger numbers with more than ten digits are printed in scientific notation only if they can be presented more accurately in scientific notation than in the normal form.*

This point is very important. Consider an example that we've tried before:

```
>PRINT 1/3
.3333333333
>□
```

We know that .333333333... is a repeating decimal that goes on infinitely. Why, then, display show the result in normal form? The answer is that .3333333333 is more accurate than 3.3333E-1; that is, more *significant digits* (digits that reflect the actual mathematical value of the number) can be shown in normal form than in scientific notation.

Scientific notation is just a "shorthand" method for writing long number, whether they are very large or very small quantities. It allows the computer to handle, in the most accurate form possible, numbers that otherwise could not be adequately displayed in the ten-digit form.

### Entering Numbers in Scientific Notation

Up to this point, we've entered numbers only in the normal decimal form. It is also possible, however, to enter numbers in scientific notation. Try this example:

```
>PRINT 1.23456E10
1.23456E=10
>□
```

Notice that, unless you enter a minus sign before the mantissa and/or the exponent, these are assumed to be positive.

```
>PRINT 2.574E13
2.574E+13
>
PRINT -5.5E-11
-5.5E
>□
```

If you enter a number in scientific notation, but the computer can show it in normal form, it will do so. Try this:

```
>PRINT 5.555E3
5555
>□
```

Whenever you are using extremely large or small numbers in a computation, entering the numbers in scientific notation can be very handy.

## Exponentiation

In the previous section we talked about *exponents* and *powers* of 10. Now we need to discuss some of the “higher math” capabilities of your computer, specifically, *powers* and *roots*.

### *Powers*

Quite often in mathematical calculations, we must raise some number to a *power*, such as

$8^3$  (or  $8 \times 8 \times 8$ )  
 $25^2$  (or  $25 \times 25$ )

To perform exponentiation (raising a number to a power) on the computer, we do this:

```
>PRINT ^3
512
>PRINT 25^2
625
>□
```

The *exponentiation symbol* (^) tells the computer that the number that follows is a power.

Let's say that we have this mathematical expression to evaluate:

$$y+x^3$$

We want to find all the values for y where x equals 1 through 10. So we enter this short program:

```
NEW
10 CALL CLEAR
20 FOR X=1 TO 10
30 Y=X^3
40 PRINT "Y="; Y
50 NEXT X
60 END
```

When we run the program, we'll see the following values for y:

```

Y= 1
Y= 8
Y= 27
Y= 64
Y= 125
Y= 216
Y= 343
Y= 512
Y= 729
Y= 1000

```

The computer completes the program for us very quickly! We have the values we need and can go on to other computations.

## Roots

Finding a *root* of a number is another very common mathematical problem. The *square root* is one we've all heard of — and probably used — at some point in our educations. Since many, many calculations call for square roots, this function is built into the computer:

```

>LET A=SQR (4)

>PRINT A
2

>PRINT SQR (16)
4

>□

```

The letters SQR stand for “square root of” and instruct the computer to find the square root of the number or expression contained within the parentheses.

Other roots must be computed by using a form of exponentiation. Computing a root of a number is the same function as raising the number to a power which is the reciprocal of the root: that is,

<sup>3</sup> 125 is the same as  $125^{(1/3)}$

Try this example:

```

>PRINT 125^
5.

>□

```



We can, however, demonstrate the presence of these internal “computational” digits. Let’s use the same problem we performed earlier.

```
>LET A=25^(1/3)
>PRINT A
 2.924017738
>PRINT A^3
 25.
>□
```

The “memory box” labeled A retains all the internal digits as well as the rounded result shown on the screen. Therefore, with the greater accuracy provided by the internal digits, we get back our original 25 when we raise A to the power of 3.

One special note of caution: Your computer will give you an error message if you try to raise a negative number to a fractional power; therefore, you cannot use the exponentiation routine to find roots of a negative number *without* taking other steps. See the Sign (SGN) and Absolute Value (ABS) functions in the BASIC Reference section of the *User’s Reference Guide*.

## Order of Operations

In Chapter 3 we discussed the order the computer follows to complete problems involving multiplication, division, addition, and subtraction. We also demonstrated that an expression within parentheses is evaluated before the rest of the problem is solved. The order of operations, then, was listed as:

- (1) Complete everything inside parentheses.
- (2) Complete multiplication and division.
- (3) Complete addition and subtraction.

Now we need to add another level to this order. Exponentiation (raising a number to a power or finding a root of a number) is performed before any other mathematical operation. So our new order becomes:

- (1) Complete parenthetical expressions.
- (2) Complete exponentiation.
- (3) Complete multiplication and division.
- (4) Complete addition and subtraction.

Let’s try some examples that help to demonstrate these concepts.

First, we’ll define some variable names for the quantities we’ll be using in our calculations. Enter these lines:

```
LET A=5
LET B=2
LET C=10
LET D=4
```

Now we're ready for the calculations:

```

>PRINT B*C^B
 200

>PRINT A+B*C^B
 205

>PRINT ((A+B)*C)^B/D
 1225

>PRINT

```

Here's the order the computer followed in each of these examples:

|                       |                                                                         |
|-----------------------|-------------------------------------------------------------------------|
| <i>First problem</i>  | $10^2=100$<br>$2 \times 100 = 200$                                      |
| <i>Second Problem</i> | $10^2=100$<br>$2 \times 100=200$<br>$5+200=205$                         |
| <i>Third problem</i>  | $5 \times 2=7$<br>$7 \times 10=70$<br>$70^2=4900$<br>$4900 \div 4=1225$ |

Notice that this last problem utilized two sets of parentheses, one within the other. In this situation the computer evaluates the innermost set of parentheses first.

As you saw when we discussed the roots of numbers, the exponent of a number can also be a numeric expression enclosed in parentheses. Let's try a few more examples, using the values already stored in the computer's memory.

```

>PRINT ((A+B)*(A+B))^(B/D)
 7.

>PRINT B^(D/B)+A*C
 54.

>□

```

The first problem essentially squared the number 7 and then took the square root of the result:

```

(A+B)=5+2=7
(A+B)*(A+B)=7x7=49
B/D=2÷=.5
49^.5= 49=7

```

The second problem is solved like this:

$$\begin{aligned} D/B &= 4 \div 2 = 4 \\ A * C &= 5 \times 10 = 50 \\ 4 + 50 &= 54 \end{aligned}$$

The following program not only demonstrates the computational power of your computer, but also plays a scale for you! The relationship between the frequencies of notes in the tempered scale can be algebraically expressed as

$$y = xk^n$$

where  $x$  = the frequency of the first note of the scale,  
 $k$  = a constant,  $^{12} 2$ ,  
 $n$  = the number of half-steps between note  $x$  and note  $y$   
 $y$  = the frequency of the next note you want to play

There are twelve notes in the tempered scale, and between each note and the next is one half-step. The following program, starting with a frequency of 440 (A above middle C on a piano keyboard), calculates and plays each note in the scale:

```

20 X=440
30 K=2^(1/12)
40 CALL SOUND 9200, X, 2)
50 FOR N=1 TO 12
60 Y=K*K^N
70 CALL SOUND (200, Y, 2)
890 NEXT N
90 END

```

Run the program and listen to the music!

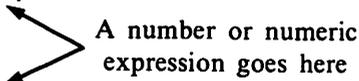
### Other Mathematical Functions

Several other mathematical functions, in addition to those we've already covered, are available in TI BASIC. We won't discuss these in detail, but we want to list some of them for you, because they can be a great help in performing mathematics with your computer.

#### *Trigonometric Functions*

These trigonometric functions are available:

**SIN ( )** Finds the *sine* of the number of numeric expression enclosed in parentheses.



**COS ( )** Finds the *cosine* of the number or numeric expression enclosed in parentheses.

**TAN ( )** Finds the *tangent* of the number or numeric expression enclosed in parentheses.

**ATN ( )** Finds the *arctangent* of the number or numeric expression enclosed in parentheses.

*Note:* All trigonometric functions are performed by the computer in *radians*, rather than *degrees*. Therefore, if your data is measured in degrees, you'll need to convert the measurement to radians before using it with the function. (To convert an angle from degrees to radians, multiply by  $\pi$ .)

*Logarithms*

The computer calculates the *natural log* and *natural antilog* (based on  $e=2.718281828$ ) of a number:

-  A number or numeric expression goes here  
**LOG ( )** Computes the natural logarithm of the number or numeric expression enclosed in parentheses.  
**EXP ( )** Computes the natural antilogarithm of the number or numeric expression enclosed in parentheses.

To convert the natural logarithm of a number to the *common log* of the number, simply divide natural log by the natural log of 10. For example, if you want to find the common log of 3, you would use this procedure:

```

>A=LOG (3) /LOG(10)
>PRINT A
.4771212547 ← Common log of 3
>□

```

*Absolute Value*

Calculations often require the use of the absolute value of a number. This has the effect of making the number positive, regardless of its sign. Here's how to instruct the computer to find and utilize the absolute value of a number:

-  A number or numeric expression goes here  
**ABS ( )** Finds the absolute value of the number or numeric expression in parentheses.
- There are other mathematical functions available, and you'll find them listed and discussed under Functions in the BASIC Reference section of the *User's Reference Guide*. The functions we've illustrated here, however, should help you discover many ways to use your computer as a computational tool.

# Appendix E

## 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

### CANT CONTINUE

1. CONTINUE was entered with no previous breakpoint or program was edited since a breakpoint was taken.

### CANT 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 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. Mis-matched 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
2. Parameter in DEF statement is not a valid variable name

#### *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) +, 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: CANT 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 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

##### JOYST, KEY

9. *Key-unit* out of range in JOYST or KEY statement

##### ON

10. *Numeric expression* indexing *line-number* is out of range

**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 large 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

**CANT 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

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
6. 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

#### *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, LOAD, 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

**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 reentered.
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 the 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 a 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           |

The second digit (Y) indicates what kind of error occurred.

| <i>Y Value</i> | <i>Error Type</i>     |
|----------------|-----------------------|
| 0              | Device name not found |
| 3              | Illegal operation     |
| 6              | Device error          |

1. Invalid device or file name in DELETE, LIST, OLD, or SAVE command
2. Not enough memoery to allocate an Input-Output buffer
3. This error can occur during file processing if an accessory device is accidentally disconnected while the program is running

**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

# Index

- Addition, 15
- Animation, 89, 209
  - flashing letters, 90
  - flashing color squares, 91
  - moving color squares, 92
  - using CALL CHAR, 209
  - using CALL SOUND, 209
- Arrays, 162, 170, 180
  - labelling elements, 181
  - multi-dimensional variables, 183
  - one-dimensional, 164
  - string arrays, 188
  - three-dimensional, 189
  - two-dimensional, 180, 184
  - using DATA statements, 182
- ASCII codes, 239
  - finding a character, 242
  - table, 241
- ASC statement, 240
  - ordering words, 244
- BASIC, 1
- Calculating commissions, 172
  - program, 173
  - running the program, 175
- CALL CHAR statement, 202, 211
  - block figure, 216
  - shorthand codes, 214
  - undefined codes, 218
  - with color, 203
- CALL CLEAR, 5, 48
- CALL COLOR statement, 68
  - background color, 70
  - character set codes, 69
  - color combinations, 72
  - foreground color, 70
  - using INPUT, 71
- CALL HCHAR, 25, 70, 272
  - horizontal positioning, 31
  - with CALL COLOR, 70, 72
- CALL KEY, 210
- CALL SCREEN, 201
- CALL SOUND statement, 20, 60, 66, 127, 278
  - duration, 21, 61
  - frequency, 21, 61
  - format, 21
  - loudness, 21, 61
  - noise, 24
  - two tones, 22
- CALL VCHAR statement, 25, 272
  - positioning a character, 27
  - vertical positioning, 29
- Character pattern program, 154, 157
  - flow chart, 156
- Color bar program, 126
- Color organ, 166, 168, 183, 186
- Command, 39
- Computer music, 129
- Correcting mistakes, 8
- Counting, 64
- Cursor, 4
- DATA statement, 138
  - error message, 139
  - more than one, 140
  - strings, 144
  - using IF-THEN, 139
  - using ON-GO TO, 142
- Delay subroutine, 269
- Dice simulation, 111, 277
- DIM statement, 163
- Division, 15
- Display, 2
- Editing, 253
  - deleting a line, 259
  - ignoring changes, 260
  - more than one line, 256
  - one line, 255
  - summary of commands, 264
- END statement, 39
- ENTER key, 5, 8, 37
- Errors, 40
  - typing, 40
- Error messages, 8, 73
  - conditions, 74
- Expression, 15
- Flow chart, 60, 62, 87
- FOR-NEXT statement, 80, 206
  - delay loop, 83
  - error conditions, 93
  - general form, 81
  - "nested" loops, 84
  - trace, 82
  - with CALL HCHAR, 83
  - with CALL VCHAR, 85
- GIANT program, 221
- GO TO statement, 58, 63
  - musical scale, 66
  - with CALL COLOR, 68
  - with CALL SOUND, 60
- GOSUB statement, 86
  - time delay, 91
- Graphics, 25
- Greatest integer function, 102
- Hertz, 20
- IF-THEN statement, 119, 130
  - comparison with IF-THEN-ELSE, 131

- IF-THEN (continued)
  - error conditions, 131
  - general form, 119, 130
  - relationships, 121
  - trace, 121
- Immediate mode, 6, 25, 30, 36, 153
- INPUT statement, 45, 206
  - assigning string variables, 51
  - strings, 47, 51
- INT function, 99
  - general form, 100
  - using negative numbers, 101
- Keyboard, 2
- LEN statement, 228, 234
  - comparing two strings, 229
- LET statement, 11, 53
  - string assignment statement, 53
- Line number, 38, 42, 261
- LIST command, 41, 44, 74
  - by line number, 44
- Loops, 58, 62, 80
  - counter, 85
  - IF-THEN, 120
  - IF-THEN-ELSE, 130
- Memory, 2
- Multiplication, 15
- Musical interlude, 129, 210
- Music Subroutine, 278
- Nested FOR-NEXT, 88
- NEW command, 39
- NUM command, 261
- Number Guessing program, 123
  - variation, 132
  - with ON-GO TO, 136
- Numeric variables, 50
- ON-GOSUB statement, 133
- ON-GO TO statement, 131
- Ordering words, 243
- POS statement, 243
- PRINT statement, 6, 12
  - colon spacing, 149, 150
  - comma spacing, 15, 145, 150
  - “empty” line, 61
  - patterns, 154, 205, 208
  - quotation marks, 7
  - semicolon spacing, 13, 16, 146, 150
  - standard positions, 147
  - strings, 13, 49
  - string variables, 148
- Program, 36
- Prompt, 4, 42
- Random notes, 127
- RANDOMIZE, 106, 124, 125
  - examples, 109
- READ statement, 138
  - list, 143
  - using ON-GO TO, 142
  - using string DATA, 144
- REMARK statement, 98
- RES command, 262
- RESTORE statement, 143
- RETURN statement, 86
- RND function, 103, 107
  - error conditions, 114
  - examples, 109
  - used with VCHAR, 112
- RUN command, 39, 42
- Screen coordinates, 26
- SEG statement, 231, 234
  - with PRINT, 232
- SHIFT key, 9, 59
- Simulations, 103
  - two dice, 111
- Statement, 5, 39
- String, 7, 11, 49
  - comparing, 245
  - concatenating, 236
  - finding length, 228
  - removing sets of characters, 276
  - removing spaces, 274
  - searching, 243
  - selecting substrings, 231
  - using numeric values, 247
- String variables, 49
  - assignment statement, 53
  - definition, 50
- Subroutines, 86
  - delay, 269
  - music maker, 278
  - redefining characters, 272
  - sound and color, 273
  - using borders, 271
  - using VCHAR/HCHAR, 272
- Subscripted variables, 162
- Subtraction, 15
- TAB function, 150
- Tone guessing program, 125, 165
- Trace, 63, 82, 121
- Undefined character codes, 218
- User’s Reference Guide, 3
- Variable, 11
  - numeric, 11

## INTRODUCTION TO TI BASIC

*Don Inman, Ramon Zamora, and Bob Albrecht*

Here's a comprehensive work, written by three of the foremost microcomputer programming experts in the country, that will teach you all about BASIC for use with the Texas Instruments Home Computer. Even if you've never worked with a computer, you can now teach yourself how to use, program, and enjoy the TI Home Computer with this entertaining and easy-to-read work. The authors have carefully constructed it so that you will soon be writing BASIC programs and exploiting all of the excellent features of the TI machines. Its 14 chapters and Appendices cover all of the essential programming statements and machine features.

### *Other books of interest . . .*

#### **BASIC BASIC: An introduction to Computer Programming in BASIC Language, Second Edition**

*James S. Coan*

"... an excellent introduction to the use of BASIC . . . clearly written and well-organized." *Computing Reviews*. "It is a well-written book . . . there are many good examples, complete with results." *Computer World*. #5106-9, paper, 288 pages

#### **BASIC WITH STYLE: Programming Proverbs**

*Paul Nagin and Henry F. Ledgard*

Covers structured BASIC programming. Each proverb is accompanied by discussion, explanations, and sample programs demonstrating the techniques. #5115-8, paper, 144 pages

#### **BASIC COMPUTER PROGRAMS FOR THE HOME**

*Charles Sternberg*

Over 75 application programs for the home. Each program is documented with a description of its functions and operation, a listing of the BASIC program, a symbol table, sample data, and one or more output samples. #5154-9, paper, 336 pages



HAYDEN BOOK COMPANY, INC.  
Hasbrouck Heights, New Jersey