Cover Copy: A key to the more progressive capabilities
               of programming in BASIC.

(title page)

Basic Computer 99/2

Book 3: Advanced BASIC Programming

Book 3: Advanced BASIC Programming

Contents

Introduction

If you have worked your way through Book 2, you have learned a lot about your computer and its language. You have been getting comfortable with the keyboard and familiar with TI-99/2 BASIC while sampling the capabilities and versatility of your computer. You have also learned quite a bit about the fundamentals of programming in TI-99/2 BASIC.

Now that you are ready for a more advanced and serious study of programming, there are a few terms and expressions that you should learn; they are a part of any serious programmer's vocabulary. These terms are simply more precise and accurate ways to discuss and describe computer programming. The first few sections of Book 3 serve to review many of the concepts you have already studied and to introduce these new terms and expressions to you. Then you are introduced to some of the more advanced capabilities of your computer.

You should enter and run all of the program examples in Book 3. Feel free to use all of the editing techniques you have learned in Book 2 to change and experiment with these examples. Refer to Book 4 if you have questions about a specific command, statement, or function.

More about Variables and Assignment Statements

As mentioned in Book 2, a variable is a name assigned to a numeric value or a sequence of characters (character string). There are two types of variables, numeric and string. Both numeric and string variables can be from 1 up to 15 characters long. Variable names must begin with either a letter, an "at" sign (@), a left bracket ([), a right bracket (]), a back slash ( ), or an underline (_). The only characters allowed in a variable name after the first character are letters, numbers, an "at" sign (@), and an underline (_). The exception to this is the dollar sign ($), which must be the last character in a string variable name. The dollar sign character may not be used in any other place in a variable name.

Values assigned to numeric or string variables are called constants. Constants are either numeric or string. The following rules must be observed with constants:

Numeric Constants:

!o! Commas and spaces cannot appear within a number.

!o! Numbers can be expressed in exponential notation, which is a special format of scientific notation in which the number 10 is replaced by the letter E; there are no spaces within the expression. Exponential notation consists of the mantissa or base number (preceded by the minus sign if negative), followed by the letter E, followed by the power of 10 (preceded by a minus sign if negative). (8.4×10^{-6}, or .000084, is the same as 8.4E-6.)

!o! Negative numbers must be preceded by a minus sign; a plus sign preceding a positive number is optional.

!o! TI-99/2 BASIC displays up to 10 digits of a number; numbers with more digits are displayed in exponential notation.

!o! Exponential notation enables you to enter and evaluate calculations with numbers of magnitudes as small as ±1E-128 or as large as ±9.9999999999999E127.

!o! Zero is a valid numeric constant.

String Constants:

!o! String constants are usually enclosed in quotation marks. The quotation marks may be omitted when a string constant is used in a DATA statement or entered to an INPUT statement.

!o! String constants include characters such as letters, numerals, spaces, and symbols; spaces within a string constant are counted as characters of that string.

!o! All characters on the keyboard that can be displayed can be used in a string.

!o! The length of a string is limited by the length of the input line (112 characters or four lines on the screen).

As discussed in Book 2, numeric and string variables are assigned values with the assignment statement. The assignment statement consists of the optional word LET, a variable, an equals sign, and a numeric or string constant. The constant to the right of the equal sign is assigned to the variable to the left of the equals sign, as shown below.

```
LET K=50
A=3.4
N$="NAME"
```

String expressions can be joined together by using an ampersand (&). This operation is called concatenation.

```
>W$="WHAT IS YOUR "
>N$="NAME?"
>Z$=W$&N$
>PRINT Z$
 WHAT IS YOUR NAME?
```

The INPUT statement also can be used to assign constants to variables. The INPUT statement causes the computer to stop the program, display either a question mark or a prompting message, and wait for a value to be entered from the keyboard. The following rules must be followed when you use the INPUT statement.

- !o! Variables in INPUT statements must be separated by commas; for example: INPUT X,Y,Z,C$
- !o! The values or data entered must correspond in number and type to the variables listed in the INPUT statement. In the previous example, three numeric values followed by one string value must be entered to the INPUT statement.
- !o! Strings entered in response to an INPUT statement may be enclosed in quotes. Strings containing commas, quotation marks, or leading or trailing spaces must be enclosed in quotes.

READ and DATA statements can also be used to assign values to variables within a program. READ and DATA statements are discussed later.

In the program on the right, A is the numeric variable. B$ is the string variable. Note that the numeric variable A in line 100 is assigned the value 10, which is a numeric constant, because it represents a number. "EASTER IS COMING" is a string constant.

```
NEW
>100 A=10
>110 B$="EASTER IS COMING"
>120 PRINT A
>130 PRINT B$
>RUN
  10
 EASTER IS COMING

  ** DONE **
```

This program example shows the use of INPUT statements, complete with prompting messages, to assign string constant values to string variables. Note that numerals are assigned as a value to the string variable A$ in line 110. RUN the program, entering your name and age at the appropriate prompts.

```
NEW
>100 INPUT "MY NAME IS ":N$
>110 INPUT "AGE? ":A$
>120 PRINT N$;A$
>130 GOTO 100
>RUN
 MY NAME IS ANN
 AGE? 18
 ANN 18

 ** DONE **
```

Unconditional Branching--Using GOTO and ON GOTO

Many times in a program you need to alter the flow of the program by directing the computer to a statement other than the next statement in sequence. In TI-99/2 BASIC, one way to do this is by means of unconditional branching statements.

The GOTO statement is used to direct the computer unconditionally to another statement. When the GOTO statement is executed, the computer always goes to the statement specified by the line number in the GOTO statement.

```
>100 INPUT X
>110 PRINT X
>120 GOTO 100
```

When the above program is executed, line 100 prompts for a value for X. After a value is entered, the computer prints the value and jumps back to line 100, as indicated by the GOTO statement. This jump (or transfer of control) is called an unconditional branch.

The above program is an example of an endless loop; the computer asks for and prints another value of X indefinitely. The BREAK or CLEAR key must be pressed to stop the program.

The ON GOTO statement, a GOTO statement with more options, has the following format.

ON variable GOTO line-list

This statement, like the GOTO statement, also can be considered an unconditional branching statement because it always transfers control to another line. However, ON GOTO transfers control to a selected line from the line-list. The line selected depends on the value of the variable.

When this statement is executed, the variable after the word ON is rounded to the nearest integer. If the value is 1, the computer branches to the first number in the line-list. If the value is 2, it branches to the second line number, and so on. If the value is less than one or greater than the number of line numbers specified (for example, the value is 4 but only three line numbers are listed), the program stops and an error message is displayed.

```
>NEW

>100 REM HOW MANY GIFTS ON
 THE 12 DAYS OF CHRISTMAS?
>110 GIFTS=0
>120 DAYS=1
>130 COUNT=0
>140 COUNT=COUNT+1
>150 GIFTS=GIFTS+1
>160 IF COUNT=DAYS THEN 180
>170 GOTO 140
>180 DAYS=DAYS+1
>190 IF DAYS<=12 THEN 130
>200 PRINT "TOTAL NUMBER OF G
 IFTS IS";GIFTS
>RUN
 TOTAL NUMBER OF GIFTS IS 78

 ** DONE **
```

In the example below, the INPUT statement prompts for a number. Enter and run the program, entering the numbers 1, 2, and 3 respectively to see the results of each.

```
>NEW

>100 INPUT "ENTER A NUMBER FR
 OM 1 TO 3 :":N
>110 IF (N<1)+(N>3) THEN 100
>120 ON N GOTO 130,150,170
>130 PRINT "NOW AT 130"::
>140 GOTO 100
>150 PRINT "NOW AT 150"::
>160 GOTO 100
>170 PRINT "NOW AT 170":"THE
 END"
```

Conditional Branching--IF THEN

Sometimes the transfer of control to another portion of the program depends upon certain conditions; thus it is called _conditional branching_. The IF THEN statement tells the computer to branch to another line if a specified condition is met. The format is as follows:

IF _condition_ THEN _line-number_

The IF THEN statement is used when a decision is to be made by the computer. If the condition is true, the computer branches to the specified line number. If the condition is false, the computer does not branch to the specified line but simply continues to the next line in sequence.

The condition used in an IF THEN statement is usually a _relational expression_, which has a value of true or false. A relational expression (for example, X=5) compares two numeric expressions or two string expressions. The mathematical symbol between the two elements in a relational expression must be one of the following _relational operators_.

=	Equal to
<>	Not equal to
<	Less than
<=	Less than or equal to
>	Greater than
>=	Greater than or equal to

Some examples of relational expressions and their values are:

Relational Expression	Value
5=11	false
N>2*7	true only if N is greater than 14
X+Y<A+B	true only if the sum X and Y is less than the sum of A and B
"THIS"="THAT"	false
"ABCD"<"ABC"	false

Enter the program on the right. When a digit is entered from the keyboard, the IF statement determines whether the digit is zero. If the digit is zero (the condition is true), line 140 is executed. If the digit is nonzero (the condition is false), line 120 is executed.

The IF THEN statement can also use the ELSE option. The general format for this statement is shown below.

IF _condition_ THEN _line-number1_ ELSE _line-number2_

The ELSE option enables you to branch to a line other than the next line in sequence if the condition is false. The program on the right, which finds the largest value in a set of numbers, uses an IF THEN ELSE statement at line 160.

The program on the right contains IF THEN statements at lines 140 and 200. At these points, the computer must decide if the problem has been answered correctly. If so, the program branches to a line that prints a reward message. If not, the program branches to a line that prints a prompt to try again. Then the problem is presented again by branching back to the appropriate line number with a GOTO statement.

```
NEW

>100 PRINT "MATH PROBLEMS"
>110 PRINT
>120 PRINT "2 TIMES 2? "
>130 INPUT A
>140 IF A=4 THEN 170
>150 PRINT "WRONG, TRY AGAIN"
>160 GOTO 120
>170 PRINT "VERY GOOD"
>180 INPUT B
>190 PRINT "6 TIMES 2? "
>200 IF B=12 THEN 230
>210 PRINT "SORRY, TRY AGAIN"
>220 GOTO 190
>230 PRINT "RIGHT"
>RUN
```

If you enter 4 for the first input prompt and 12 for the second, you get the result on the right. Run the program again and see what happens when you answer the problems incorrectly.

```
 MATH PROBLEMS
 2 TIMES 2?
 ?4
 VERY GOOD
 ?12
 RIGHT

 ** DONE **
```

```
>NEW

>100 INPUT  "ENTER A DIGIT: ":D
>110 IF D=0 THEN 140
>120 PRINT "NONZERO DIGIT ENTERED"
>130 GOTO 100
>140 PRINT "ZERO ENTERED"
>150 GOTO 100
```

Lines 100 and 110 prompt for the number of values to be entered (TOTAL) and the first value (NUMBRE), respectively. The value of NUMBRE is assigned to the variable LARGE and TOTAL is reduced by 1. If all the values have been entered, the value of LARGE is printed. Otherwise, the INPUT statement at line 150 prompts for another value, which is then compared to the current LARGE. If the value of LARGE is still the larger, the program branches to line 130. If the value of LARGE is less than the value entered, LARGE is assigned the new value in line 120.

```
>NEW

>100 INPUT "NUMBER OF VALUES?: ":TOTAL
>110 INPUT "ENTER VALUE: ":NUMBRE
>120 LARGE=NUMBRE
>130 TOTAL=TOTAL-1
>140 IF TOTAL<=0 THEN 170
>150 INPUT "ENTER VALUE: ":NUMBRE
>160 IF LARGE>NUMBRE THEN 130 ELSE 120
>170 PRINT "LARGEST VALUE ENTERED=";LARGE
```

The IF THEN statement can also be used with numeric expressions. A numeric expression is evaluated, and if it has any nonzero value, the expression is considered true. The expression is considered false if it has a zero value. In the example

```
IF N THEN 120
```

the value of N is compared with zero. If N is equal to zero, the condition is false. If N has any nonzero value, the condition is true.

The IF THEN statement also can be used to determine if more than one relational expression is true or false. This testing of multiple expressions involves using the logical AND or the logical OR. The logical AND requires that all the relational expressions included in the condition be true for the condition to be true. The logical OR requires that at least one of the relational expressions in the condition be true.

A logical AND can be implemented on the TI-99/2 Computer by using the multiplication operator (*) between relational expressions as in the example below.

```
IF (20>2*7)*(2<5) THEN 120
```

The relational expression 20>2*7 is true, which means it has a nonzero value. The relational expression 2<5 is also true. Because neither expression has a value of zero, the product of these two true expressions is a nonzero value. Therefore, program control is transferred to line 120.

Note that the statement

```
IF (20>2*7)*(2>5) THEN 120
```

is false because one of its expressions is false (has a value of zero).

The statement

```
IF (N>2*7)*(A>5) THEN 120
```

compares the values of variables using the previous IF THEN statement. If N has a value greater than 14, the first relational expression is true (has a nonzero value). If A has a value greater than 5, the second relational expression is true. If either expression is false, the relational expression has a value of zero, which makes the condition false (have a zero value).

A logical OR is implemented on the Basic Computer 99/2 by using the addition operator (+) between relational expressions. A logical OR checks whether either expression is true (has a nonzero value). If so, the condition is considered true. In the example

```
IF (20>2*7)+(2>5) THEN 120
```

the first expression is true (has a nonzero value) and the second expression is false (has a zero value). The sum of these expressions results in a nonzero value, which is considered true.

More about FOR-NEXT Loops

As discussed in Book 2, FOR and NEXT statements are useful for specifying how many times a loop is to be performed. The loop begins with a FOR TO statement as shown in the example on the right.

When you specify the number of times the loop is to be performed in line 100, you are setting conditions that control when and where the computer branches. This FOR TO statement sets up a loop that is executed 100 times. 1 is called the first or _initial value_ and 100 is the _limit_. A acts as a _counter_ for the loop and is called a _control variable_.

At line 100, A is assigned a value of 1. Line 110 prints the current value of A. The NEXT statement at line 120 marks the end of the FOR-NEXT loop. The NEXT statement increments A by 1 and checks whether the value of A is greater then the limit. If the value of A is not greater than the limit, control is transferred back to line 110. Each time through the loop the value of A is increased by 1 until the maximum value of A (100) is reached.

The FOR TO statement has an option (not covered in Book 2) that enables you to increment the value of the control variable by any integer you choose. This is done by including the word STEP and the integer to be added. Modify line 100 as shown on the right.

When the loop begins, the variable A is equal to 1 (the initial value). The value of A is printed. The NEXT statement increases A by the value specified by STEP (in this case, 10). The loop is repeated until the value of A has reached the limit. If the STEP option is not used, the computer automatically increments the value of the control variable by +1.

You can also cause the control variable to decrease instead of increase by including the STEP option with a negative increment. As before, the control variable is assigned the initial value. The second time through the loop the control variable is decreased when the negative increment is added to the initial value. The control variable is decreased by the increment until the control variable equals the limit.

In the previous example, A is 10 the first time through the loop. As the loop is performed, the value of A is 8, 6, 4, 2, and 0. When A equals 0, the program ends.

As discussed in Book 2, you can create loops within loops by using multiple FOR and NEXT statements. The program on the right has two FOR-NEXT loops. The first loop starts at line 100 and ends at line 140. Within this loop, there is another loop that begins at line 110 and ends at line 130. This loop, called a _nested loop_, is embedded in the first loop. When using a nested loop, you must end it with a NEXT statement before you end the outer loop (NEXT B must come before NEXT A). Different control variables must be used for each loop.

```
>NEW

>100 FOR A=1 TO 100
>110 PRINT A
>120 NEXT A


>100 FOR A=1 TO 100 STEP 10
>110 PRINT A
>120 NEXT A


>100 FOR A=10 TO 0 STEP -2
>110 PRINT A
>120 NEXT A


>NEW

>100 FOR A=1 TO 10 STEP 2
>110 FOR B=2 TO 11 STEP 3
>120 PRINT A;B
>130 NEXT B
>140 NEXT A
```

There are 3 loops in the program on the right. Two are nested inside the first one. Note that the three control variables have different names (X, Y, Z).

```
>NEW

>100 FOR X=1 TO 4 STEP 3
>110 FOR Y=5 TO 10 STEP 5
>120 FOR Z=30 TO 40 STEP 10
>130 PRINT X;Y;Z
>140 NEXT Z
>150 NEXT Y
>160 NEXT X
>RUN
  1  5  30
  1  5  40
  1  10  30
  1  10  40
  4  5  30
  4  5  40
  4  10  30
  4  10  40

 ** DONE **
```

X=1 (initial value) for all values of Z and Y. Then X is incremented by 3.

Using NUM (NUMBER) and RES (RESEQUENCE)

Thus far, you have typed your own line numbers for each program that you have entered. Your computer has a feature, however, whereby you can instruct it to automatically generate line numbers for you. This is especially convenient when you are entering longer programs.

To use this feature, just enter the command NUM (or NUMBER). The computer is now in Number Mode. Notice that the first line number generated is 100. Notice also that the space after the number is already inserted. Enter this program line.

```
     PRINT "BASIC COMPUTER"
```

When you press ENTER, notice that the next line number generated is 110. Now enter this program line.

```
     GOTO 100
```

When you press ENTER, line number 120 is generated. To leave Number Mode, just press ENTER again (without typing a program line 120). Clear the screen with CALL CLEAR and LIST the program. Notice that there is not a line 120 in the program. You can now RUN the program. Press BREAK to stop the program.

Obviously the NUM command as used above begins numbering with 100 and continues in increments of 10. If you like, you can change the initial number and the increment to any integer value (up to 32767). Enter NEW and enter the following command.

```
     NUM 3,5
```

Now enter the same program as above, changing the second line to GOTO 3. When line number 13 is generated, just press ENTER. Your program should look like this.

```
     3 PRINT "BASIC COMPUTER"
     8 GOTO 3
```

You have probably discovered that the first number after the word NUM indicates the initial line and that the second number (after the comma) indicates the increment between all the line numbers thereafter.

Now suppose you want to change the initial line and increment of the line numbers of this program. Rather than taking the time to enter the program again, you can use the RESEQUENCE command to change the line numbers. Enter the command RES and list the program. It now looks like this.

```
     100 PRINT "BASIC COMPUTER"
     110 GOTO 100
```

Notice that the line numbers start at 100 and continue in increments of 10. Notice also that the line number referenced in the GOTO statement is changed to match the new number assigned to the first line. This is done with any other branching statement also, such as IF THEN, IF THEN ELSE, or ON GOTO. Enter this command.

```
     RES 1,2
```

List the program again.

```
     1 PRINT "BASIC COMPUTER"
     3 GOTO 1
```

As you can see, the numbers following RES indicate the initial line and the increment as they do with the NUM command. These numbers must be separated by a comma.

If you use either NUM or RES without a specified initial line or increment, the computer assumes an initial line of 100 and an increment of 10. You can also specify one value but use the assumed value for the other. This works with either NUM or RES. If, for example, you want the line numbers to start at 10 and continue in increments of 10, you can enter this command.

```
     RES 10
```

List the program again.

```
     10 PRINT "BASIC COMPUTER"
     20 GOTO 10
```

Because you have specified an initial value of 10 but have not specified an increment, the computer assigns the number 10 to the first line and assumes an increment of 10. To specify the increment but use the assumed initial line, enter the command below. Note the position of the comma.

```
     RES ,50
```

List the program again.

```
     100 PRINT "BASIC COMPUTER"
     150 GOTO 100
```

The computer assigns the assumed initial line number (100) and continues at the specified increment of 50.

Experiment with NUM and RES and use them as needed in your work with your computer. For more details on these commands, refer to Book 4.

Using READ and DATA Statements

There are three ways to assign values to variables in a BASIC program. The LET or assignment statement and the INPUT statement have been discussed. The READ and DATA statements can also be used to assign values to variables. READ and DATA are especially useful when there are many variables involved in a program.

The READ statement assigns each variable in its variable list a value from the DATA statement. The READ statement assigns the first value in a DATA statement to the first variable in the variable list. The READ statement then assigns each value in sequence in the DATA statement to each sequential variable until all the variables in the list have been assigned. If there are not enough values in a DATA statement, the READ statement assigns values from the next DATA statement. If some values in a DATA statement are not assigned, the computer remembers the first unassigned value, which will be assigned by the next READ statement.

Both the variables in the READ statement and the values in the DATA statement are numeric and/or string variables, separated by commas. Values in DATA statements must correspond to the order and type of variables specified by the READ statements. If the READ statement lists a numeric variable, a numeric constant must be in the corresponding position in the DATA statement. If the READ statement lists a string variable, a string constant must be in the corresponding position in the DATA statement. Because a numeral can be a valid string, a numeral can appear in a DATA statement where a string is required in the READ statement. If a DATA statement contains two consecutive commas, a null string (a string with no characters) is assigned.

```
READ A,B,C$
DATA 6,13,HELLO
```

In a DATA statement, leading and trailing spaces are ignored. Quotation marks are required for string constants that contain commas, quotation marks, or leading or trailing spaces. If a string constant is enclosed in quotation marks, each single quote desired within the string must be represented by a double quotation mark in the statement.

DATA statements can appear anywhere in the program. Because DATA statements merely store values to be read by READ statements, DATA statements are not executed; the computer simply continues to the next statement. It is good programming practice to group DATA statements together either at the beginning or at the end of the program.

The RESTORE statement is used to alter this sequential assignment of values. After a RESTORE statement is executed, the next READ statement begins assigning the first value in the DATA statement specified by RESTORE.

The program on the right causes the computer to READ and PRINT both numeric and string constants. Data are stored in lines 100 and 110. Lines 120 and 130 read the data and sequentially assign the values 1, 2, 5, 3, -4, and XY to the variables A, B, C, D, E, F$. Line 140 prints the values two at a time.

```
>NEW

>100 DATA 1,2,5,3,-4
>110 DATA XY
>120 READ A,B,C
>130 READ D,E,F$
>140 PRINT A,B,C,D,E,F$
>RUN
  1            2
  5            3
 -4           XY

 ** DONE **
```

Line 100 stores data that are read and assigned to variables X and Y of line 110. Line 120 prints the values of X and Y. Try changing the position of the DATA statement. Put it after the PRINT statement. You should get the same results.

```
>NEW

>100 DATA 2,10
>110 READ X,Y
>120 PRINT X,Y
>RUN
  2            10

 ** DONE **
```

Lines 100-130 read 3 sets of data and prints their values, two to a line.

Lines 140-170 read two sets of string constants and print each set on its own line.

Lines 180-190 store numeric constants for variables B and C of line 110. Lines 200-210 store string variables for C$ of line 150.

```
>NEW

>100 FOR A=1 TO 3
>110 READ B,C
>120 PRINT B;C
>130 NEXT A
>140 FOR A=1 TO 2
>150 READ C$
>160 PRINT C$
>170 NEXT A
>180 DATA -5,0,8
>190 DATA 1,2,4
>200 DATA "HELLO, BILL"
>210 DATA "HOW ARE YOU?"
>RUN
 -5  0
  8  1
  2  4
HELLO, BILL
HOW ARE YOU?

 ** DONE **
```

```
>NEW

>100 CALL CLEAR
>110 PRINT "COMPARE PRICES OF
  10":"PRODUCTS IN TWO STORES
 "::
>120 INPUT "NUMBER OF PRODUCT
 S (1-10) ":N
>130 IF (N<1)+(N>10) THEN 120
>140 FOR S=1 TO 2
>150 PRINT "STORE";S
>160 FOR P=1 TO N
>170 READ ITEM
>180 PRINT ITEM;
>190 NEXT P
>200 PRINT ::
>210 RESTORE 240
>220 NEXT S
>230 DATA 7,10,11,27,34,61,41
 ,56,29,78
>240 DATA 9,14,19,24,28,39,55
 ,30,77
>RUN
 COMPARE PRICES OF 10
 PRODUCTS IN TWO STORES

NUMBER OF PRODUCTS (1-10) 5
STORE 1
 7  10  11  27  34

STORE 2
 9  14  19  24  28

** DONE **
```

Using Functions in TI-99/2 BASIC

The Basic Computer 99/2 has a set of built in instructions called <u>functions</u>. These built-in functions are classified as either numeric functions or string functions. Numeric functions, which perform computations on numeric expressions, can calculate routine mathematical operations much like an advanced calculator. String functions operate on character strings to do various string manipulations.

A function is always followed by an expression enclosed in parentheses. This value within parentheses is called an <u>argument</u>. An argument is the number or string on which the function operates. Arguments used with numeric functions can be single numbers, single variables, or expressions containing a mixture of variables and numbers. Arguments used with string functions can be string variables or string constants.

A function can be used in an expression in place of a variable. A function is assigned a value when the expression is evaluated. The value of a function is evaluated by the computer by performing a specialized operation on the numeric expression or string expression listed as the argument. A function can never appear on the left side of an equals sign.

Refer to Book 4 for information on other numeric functions that can be used with your computer.

Using Trigonometric Functions--SIN, COS, TAN, and ATN

TI-99/2 BASIC has four trigonometric functions. SIN(X), COS(X), and TAN(X) calculate the sine, cosine, and tangent, respectively, of X, where X is an angle measured in radians. When an expression containing one of these functions is evaluated, the computer uses the numeric expression following the function name to obtain the value of the argument; calculates the sine, cosine, or tangent of that value; and uses that value for the function name.

For example, at line 160 the expression 5*COS(X) is evaluated as shown below.

```
>150 X=20
>160 L=5*COS(X)
```

Find the cosine of 20 radians (.4080820618)
Multiply the value of the function by 5 (2.040410309)

There are 360 degrees or 2Ü radians in a circle. If you prefer to work with angles measured in degrees, you can convert degrees to radians as follows.

$$\underline{\text{degrees}} \times \frac{2\text{Ü}}{360} \text{ or } \underline{\text{degrees}} \times \frac{\text{Ü}}{180} \text{ or } \underline{\text{degrees}} \times .01745329251994$$

The ATN(X) function (arctangent) calculates the angle in radians whose tangent is X. To convert the radians to degrees, multiply the function by 180/Ü or 57.2957795131.

The program on the right computes the sine, cosine, and tangent of 45 degrees.

```
>NEW

>100 X=45*0.01745329251994
>110 PRINT SIN(X):COS(X):TAN(
 X)
>RUN
  .7071067812
  .7071067812
  1.

 ** DONE **
```

This program converts the ATN of 1 back to degrees.

```
>NEW

>100 A=57.2957795131*ATN(1)
>110 PRINT A
>RUN
  45.

 ** DONE **
```

Using EXP, LOG, and SQR Functions

Other numeric functions include the EXP, LOG, and SQR functions.

The EXP(X) function returns the value of e^X, where e=2.718281828. The exponential function is the inverse function of the natural logarithm function LOG. Thus, X=EXP(LOG(X)).

The LOG(X) function returns the natural logarithm of the number specified by the argument X. The argument X must always be greater than 0.

The SQR(X) function calculates the square root of the argument X. SQR(X) is equivalent to X^(1/2). The value of X must be greater than 0.

The program on the right calculates the EXP, LOG, and SQR of the arguments in parentheses and prints the answers.

```
>NEW

>100 A=EXP(4)
>110 B=LOG(A)
>120 C=SQR(B)
>130 PRINT A:B:C
>RUN
  54.59815003
  4
  2

 ** DONE **
```

Using String Functions--STR$, CHR$, and LEN

STR$(A) or string-number function of A converts the number specified by the argument A into a string. The argument may be any numeric expression. When a number is converted into a string, the string is a valid representation of a numeric constant with no leading or trailing spaces.

CHR$ or character function returns the character corresponding to the ASCII character code specified by the argument. If the argument is not an integer, it is rounded to an integer. If the value of the argument is between 32 and 127, a standard character is returned. For values between 128 and 159, a special graphics character is returned. The character codes for the standard character set are listed in Book 4 (BASIC Reference).

LEN or length function returns the number of characters in the string specified by the argument. The LEN of a null string is zero. A space is a character and counts as part of the length.

In the example on the right, note that leading and trailing spaces are not present in the number converted into a string.

```
>NEW

>100 A=-10.2
>110 PRINT STR$(A);"    ";A
>RUN
 -10.2    -10.2

 ** DONE **
```

In line 100, the value of A$ is defined to be equal to CHR$(37). Line 110 prints A$ or % (37 is the ASCII code for %).

```
>NEW

>100 A$=CHR$(37)
>110 PRINT A$
>RUN
 %

 ** DONE **
```

The value of B$ is "HELLO". "HELLO" has 5 characters; therefore LEN(B$) is 5.

```
>NEW

>100 B$="HELLO"
>110 PRINT LEN(B$)
>RUN
  5

 ** DONE **
```

Review

Answer the following questions to review what you have learned so far.

1. Write a BASIC program using READ and DATA statements to print the following.

```
HI.
MY NAME IS MARY.
AGE 10
```

2. Write the program lines missing at lines 110 and 140.

```
>100 FOR A=1 TO 5
>110
>120 PRINT X
>130 NEXT A
>140
>RUN
  2
  4
  6
  8
  10

 ** DONE **
```

3. Write a BASIC program that assigns a degree angle (A) and calculates its sine, cosine, and tangent. A is equal to 90 degrees.

4. Enter and run the following program. At the input prompt, enter 32. What happens? Run the program several times, entering a different character code number each time.

```
>100 INPUT A
>110 PRINT CHR$(A)
```

5. Write a BASIC program that prints the following result. Hint: Use READ and DATA statements and the CHR$ function. Refer to the ASCII Character Code chart in Book 4.

```
    +
    /
    =
    :
    ;

    ** DONE **
```

6. Fill in the blank.

a) The ___________ function returns the number of characters in the string specified by the argument.

b) _________ statements can appear anywhere in a program.

c) X^(1/2) is equivalent to __________(X).

d) _______________ is the inverse of the LOG function.

e) _________(X) calculates the angle whose tangent is X.

f) The STR$ or string-number function converts the number specified by the argument into a __________.

g) The _______ statement tells the computer to look at values in the DATA statement.

h) _______ statements alone do not cause the computer to do anything, because they merely store values to be read by the computer.

i) Three statements assign values to variables in a BASIC program; they are the _____ statement, the ______ statement, and the ______ and ______ statements.

Using Subscripted Variables

Until now, all of our programs have used only simple variables. These have consisted of the letters A-Z, the digits 0-9, and the dollar sign (used only to denote a string variable).

Let's look at a new type of variable. Enter the following into your computer:

```
      DIM A(4)
```

This statement, the DIMENSION statement, tells the computer to reserve space for four values to be assigned to the variable A. It sets the dimension in the computer's memory for this variable group. Now enter the following statements.

```
      A(1)=10
      A(2)=15
      A(3)=20
      A(4)=25
```

Each of these variables is a subscripted variable. Although they look different from anything you have seen so far, they may be used in exactly the same way as simple variables. To illustrate that this is true, enter the following:

```
      PRINT A(2),A(4)
      PRINT A(1)+A(3)
      PRINT A(2)*A(3)
      PRINT A(4)/A(1)
```

In each case, you get exactly the same result as you would have if you had substituted W for A(1), X for A(2), Y for A(3), and Z for A(4). But if both simple and subscripted variables may be used in exactly the same way, what is the difference? What is there about a subscripted variable that makes it unique?

The most obvious difference is in the way it is written. Notice that the subscripted variables all share a common name, in this case A. Had you used simple variables (W, X, Y, and Z), each variable would have a separate, unique name. This may not seem very important at first, but it proves to be extremely useful.

What is an Array?

A set of subscripted variables sharing the same name is called an array. The easiest way to picture an array is to think of it as a box divided into separate sections, or elements.

The DIMENSION statement reserves a specified number of elements for a particular array. The DIM statement tells the computer the maximum value that a subscripted variable in an array can have. You must have a DIM statement in your program before any reference to a subscripted variable from that array.

The DIM statement has the form:

```
     DIM array-name(integer)
```

The following DIM statement tells the computer that 6 is the largest subscript allowed in array X.

```
     DIM X(6)
```

If you try to use 7 as a subscript for array X, an error message is displayed. You can reserve more space than is actually needed for an array with your DIM statement. However, reserving much more memory space than is needed for the array is a wasteful programming practice because that memory cannot be used for anything else.

The box itself represents the array, and the individual elements represent the subscripted variables within the array.

Array X

X(1)	X(2)	X(3)	X(4)	X(5)	X(6)
First Element	Second Element	...	...	...	Sixth Element

The array as a whole is referenced by an array name, in this case X. From using simple variables, you know that any variable must have a unique name, one that distinguishes that variable from all others in that program.

Before you can identify any particular variable in an array, you must know two things: the name of the array and the element containing the variable. This is exactly what a subscripted variable identifies. X(1) references the first element in array X, X(2) the second element, and X(6) the sixth element. Thus, the subscript is used to tell us which element contains the variable we are interested in.

What Kinds of Expressions Can Be Used as Subscripts?

A constant, numeric variable, or numeric expression may be used as a subscript.

Constants

```
X(1)
X(5)
```

Numeric Variables Assuming you have defined A=1, B=2, and C=3, the following are permissible.

```
X(A)=X(1)
X(B)=X(2)
X(C)=X(3)
```

Any numeric variables used as a subscript must be simple variables. That is, they cannot be subscripted variables themselves.

Numeric Expressions Again, assuming A, B, and C have been defined, the following are permissible.

```
X(2+3)=X(5)
X(2+B)=X(4)
X(B*C)=X(6)
```

So far we have only shown examples of integer expressions being used as subscripts. Using our picture of an array as a box, it is easy to see that although we may have a 1st element or even a 101st element, the idea of a 1.4th element or a 6.3th element just doesn't make any sense. Therefore, the only valid subscript recognized by the computer is an integer. However, the computer accepts fractional expressions, automatically rounding the expression to obtain an integer. Enter the following:

```
    X(6)=10
    PRINT X(6.3)
     10
```

Notice that X(6.3)=X(6).

Using FOR-NEXT Loops with Arrays

Because an array is usually a sequential set of subscripted variables, every element in array X may be accessed by beginning with X(1) and incrementing the subscript by 1. This approach is ideal for use with the FOR-NEXT loop. Suppose you want to input data into the first 4 elements of array X. Enter the following program segment.

```
>NEW

>100 REM ENTERING DATA INTO AN ARRAY
>110 DIM X(4)
>120 FOR J=1 TO 4
>130 INPUT X(J)
>140 NEXT J
```

Following through the program, you can see how useful the FOR-NEXT loop becomes. On the first pass through line 120, the computer assigns a value to X(1). On the second pass, a value is assigned to X(2), and so on until a value is assigned to X(4).

The FOR-NEXT loop may be similarly used to sequentially print ar array. Suppose we now want to print the values that were input above. Enter the following program segment.

```
>150 REM PRINTING THE ARRAY
>160 FOR J=1 TO 4
>170 PRINT X(J)
>180 NEXT J
```

As you can see, anytime you are dealing with a sequential set of subscripted variables, the FOR-NEXT loop is an convenient, easy way of manipulating the array.

Why Do We Use Arrays?

Every variable in a program must be referred to by a unique variable name. When you are working with a large number of variables, providing a separate name for each can quickly make the simplest programs large and unmanageable. Also, the program must explicitly refer to each different variable name. However, by assigning data to the elements of an array, you need only use a single array name followed by a unique subscript. Additionally, because the subscript is a numerical value, it may be easily manipulated by the computer.

The following program is a simple program for computing the average test grade of a class containing N students.

```
>NEW

>100 REM TEST GRADE AVERAGES
>110 INPUT "NUMBER OF STUDENTS":N
>120 SUM=0
>130 FOR K=1 TO N
>140 INPUT "GRADE ":GRADE
>150 SUM=SUM+GRADE
>160 NEXT K
>170 AVE=SUM/N
>180 PRINT "THE AVERAGE GRADE IS ";AVE
```

At first glance, it may appear as though the variable GRADE contains N different grades. However, this is not the case. At any particular time, GRADE contains only one value. On each pass though the FOR-NEXT loop, a new value is input to the variable GRADE, replacing the previous value. Though the previous value is included in the cumulative total in SUM, it is has been lost as an individual value. For instance, once the 8th grade is replaced with the 9th, we can never know the exact value of what the the 8th grade is.

Suppose we wish to print out each grade <u>along with its deviation from the average</u>. Because we must have access to each grade after the average is computed, each grade must be assigned to a separate variable. We could assign the first student's grade to G1, the second student's grade to G2, and so on. For example, suppose there were five students.

```
>NEW

>100 REM TEST GRADE AVERAGES AND DEVIATIONS
>110 INPUT "GRADE ":G1
>120 INPUT "GRADE ":G2
>130 INPUT "GRADE ":G3
>140 INPUT "GRADE ":G4
>150 INPUT "GRADE ":G5
>160 SUM=G1+G2+G3+G4+G5
>170 AVE=SUM/5
>180 PRINT "THE AVERAGE GRADE IS ";AVE
>190 PRINT "GRADE =";G1,"DEV =";G1-AVE
>200 PRINT "GRADE =";G2,"DEV =";G2-AVE
>210 PRINT "GRADE =";G3,"DEV =";G3-AVE
>220 PRINT "GRADE =";G4,"DEV =";G4-AVE
>230 PRINT "GRADE =";G5,"DEV =";G5-AVE
```

Even with only five students, it is easy to see how large and complicated this program would become if there were forty students, or one hundred. Even a program for such a simple purpose as this would quickly get out of hand.

An array offers an ideal solution. Instead of assigning each grade to a separate variable, let's assign each grade to a subscript of an array named GRADE. The following program could easily be set up for any number of students, and it is still two lines shorter than the previous program for only five students.

```
>NEW

>100 REM TEST GRADE AVERAGES AND DEVIATIONS
>110 DIM GRADE(50)
>120 INPUT "NUMBER OF STUDENTS ":N
>130 SUM=0
>140 FOR K=1 TO N
>150 INPUT "GRADE ":GRADE(K)
>160 LET SUM=SUM+GRADE(K)
>170 NEXT K
>180 LET AVE=SUM/N
>180 PRINT "THE AVERAGE GRADE IS ";AVE
>190 FOR K=1 TO N
>200 PRINT "GRADE= ";GRADE(K),"DEV= ";GRADE(K)-AVE
>210 NEXT K
```

Searching an Array

One of the most common problems encountered in dealing with arrays is the need to search the array to determine if a particular data item is present. You may also wish to know how many times the item is present and in which of the array elements it is located.

Suppose we wish to search an array of 30 items to determine whether a given item, TARGET, is present. To begin, we must instruct the computer to compare TARGET against the item in the first element of the array, and, if the two are equal, to print the element number. Next, compare TARGET against the second element and repeat this process until every element has been checked.

Assuming that values have already been assigned to both the array X and the variable TARGET, the following program segment may be used for the search:

```
>NEW

>100 REM SEARCHING THE ARRAY
>110 FOR K=1 TO 30
>120 IF X(K)=TARGET THEN 130 ELSE 140
>130 PRINT "TARGET IS IN ELEMENT ";K
>140 NEXT K
```

On the first pass through line 120, X(1) is compared with TARGET. If the two are equal, the program continues to line 130 and the subscript number is printed. If the the two are not equal, the program passes to line 140. The FOR-NEXT loop is then incremented and TARGET is compared with X(2). This repeats until the entire array has been checked.

Using Two-dimensional Arrays

Thus far you have studied arrays that are "one-dimensional," that is, a single list of values assigned to a variable. Sometimes a one-dimensional array or a list is not suitable for storing a set of data. Some groups of data are better arranged in rows and columns (as in a table) than in a list. In TI-99/2 BASIC, an array can be expanded to two dimensions (with two subscripts) to hold the values in a table. The first subscript denotes the row where an element is located and the second subscript denotes the column where an element is located.

As a simple example, the multiplication table for the first five counting numbers is shown below. You can use a two-dimensional array to store the values in the table.

If we call the array M, you can refer to any element in the array by typing M followed by the element's row and column in parentheses. To reference the elements in the second row of the array M:

M(2,1)	refers to the element in the second row and the first column, which has a value of 2.
M(2,2)	refers to the element in the second row and the second column, which has a value of 4.
M(2,3)	refers to the element in the second row and the third column, which has a value of 6.
M(2,4)	refers to the element in the second row and the fourth column, which has a value of 8.
M(2,5)	refers to the element in the second row and the fifth column, which has a value of 10.

As with one-dimensional arrays, the subscripts here could also be variables.

Any valid operator can be used on the elements of multidimensional arrays. For example, you can use the arithmetic and the relational operators on elements of numeric arrays. You can use the concatenation operator (&) and the relational operators on elements of string arrays.

To access the elements of a two-dimensional array, you will often use nested FOR-NEXT loops. The outer FOR-NEXT loop can be used to access each row of elements and the inner loop can be used to access each element in a row.

The program on the right shows how to use a two-dimensional array to display a multiplication table (up through 49 values). The outer FOR-NEXT loop controls the row subscript. The first time through the loop, the row subscript is 1. The inner loop then increments the column subscript to define each element in the first row. Then the outer loop increments to the second row and the inner loop defines each element in that row. The two loops continue until each element in each row is defined.

Now suppose you want to search the array for the number of times a given number occurs in the table. By adding lines 210-290, you can input a number to be searched for, use two FOR-NEXT loops to compare each element in the array with the input number, and print the number of times the input number occurs in the array.

C O L U M N S

		1	2	3	4	5
R	1	1	2	3	4	5
O	2	2	4	6	8	10
W	3	3	6	9	12	15
S	4	4	8	12	16	20
	5	5	10	15	20	25

```
>NEW

>100 REM TWO-DIMENSIONAL ARRA
 Y
>110 DIM M(7,7)
>120 INPUT "ENTER NUMBER, 1-7
 : ":NUMBRE
>130 FOR ROW=1 TO NUMBRE
>140 FOR COL=1 TO NUMBRE
>150 M(ROW,COL)=ROW*COL
>160 PRINT M(ROW,COL);TAB(COL
 *4);
>170 NEXT COL
>180 PRINT
>190 PRINT
>200 NEXT ROW

>210 COUNT=0
>220 INPUT "SEARCH FOR WHAT N
 UMBER?: ":TARGET
>230 FOR ROW=1 TO NUMBRE
>240 FOR COL=1 TO NUMBRE
>250 IF M(ROW,COL)<>TARGET TH
 EN 270
>260 COUNT=COUNT+1
>270 NEXT COL
>280 NEXT ROW
>290 PRINT TARGET;"APPEARS";C
 OUNT;"TIMES"
```

Using Three-dimensional Arrays

Sometimes even two-dimensional arrays are not suitable for storing the values of certain groups of data. For example, suppose a company keeps a sales record for each of its salesmen. Each sales record contains the number of items sold and the price per unit for each item. In addition, the company needs to record the revenue each salesman produces for each item sold.

Suppose the salesmen are assigned numbers as shown below.

1	Jones
2	Smith
3	Anderson
4	Beebe

The items sold and the prices per unit are listed below.

1	shirts- 20.00
2	slacks- 24.00
3	shoes- 65.00
4	socks- 1.25

A one-dimensional array can be used to store the price of each item.

PRICE(1)	20.00
PRICE(2)	24.00
PRICE(3)	65.00
PRICE(4)	1.25

The sales records for the four salesmen are listed below.

	ITEM			
SALESMAN	1	2	3	4
1	200	250	300	500
2	300	150	375	235
3	200	560	670	346
4	245	680	456	645

To store the number of items sold by each salesman, we will use a two-dimensional array, where the first subscript denotes the salesman (1-4) and the second subscript denotes the item sold.

For each element in the table (or two-dimensional array) of the salesmen's records, we must compute and store the revenue generated by each salesman for each item. All of these revenue values can be thought of as written on a page. Because we need only one computed value for each element in the table, we need only one page. This page can be stored by adding another dimension to the array. We need a value of 1 for this third dimension. If we needed to compute and store another value, we could increase this third dimension by 1.

We can store the revenue for each item sold by each salesman as shown in the program segment on the right. We will also print the values being stored. With TI-99/2 BASIC an array can have a maximum of three dimensions, or three subscripts.

```
>NEW

>100 REM THREE-DIMENSIONAL AR
 RAY
>110 DIM PRICE(10),QUAN(10,10
 ),REV(10,10,1)
>120 INPUT "ENTER # OF ITEMS
 IN RECORD: ":ITEMS
>130 FOR ROW=1 TO ITEMS
>140 INPUT "ENTER PRICE OF IT
 EMS: ":PRICE(ROW)
>150 NEXT ROW
>160 PRINT



>170 INPUT "ENTER # OF SALESM
 EN: ":MEN
>180 PRINT
>190 FOR SMAN=1 TO MEN
>200 PRINT "SALESMAN ";SMAN;"
  RECORD"
>210 FOR COL=1 TO ITEMS
>220 INPUT "ENTER QUANTITY SO
 LD OF ALL ITEMS: ":QUAN(SMAN
 ,COL)
>230 NEXT COL
>240 PRINT
>250 NEXT SMAN



>260 FOR SMAN=1 TO MEN
>270 PRINT "RECORD FOR SALESM
 AN ";SMAN
>280 FOR COL=1 TO ITEMS
>290 REV(SMAN,COL,1)=PRICE(CO
 L)*QUAN(SMAN,COL)
>300 PRINT "$";REV(SMAN,COL,1
 );
>310 NEXT COL
>320 PRINT
>330 PRINT
>340 NEXT SMAN
```

Review--Answers are on page XX.

1. Why should arrays be used when you are processing a large number of variables?

2. The character that is enclosed in parentheses after an array name is called a ______________.

3. To access an element in an array, you must know what two things.

   ____

   ____

4. A subscript may be a ____________, a _____________, or an ____________.

5. In TI-99/2 BASIC, the maximum number of subscripts is __________.

6. Each subscripted variable in an array is called an _____________.

7. What is the difference in an array named A and an array named A$?

8. If you type A(2,3), which element in array A are you accessing?

9. What statement is used to reserve space for an array?

10. If you have entered the statement DIM A(15), can you use 20 as a subscript for the array A?

11. Write a program that accepts a digit from 1 through 5 from the keyboard and then spells that digit.

```
>NEW

>100 REM USING A SUBROUTINE
>110 INPUT "ENTER DATA IN MM/
 DD/YY FORMAT: ":DATE$
>120 A$="BEGINNER'S BASIC"
>130 PCOUNT=143
>140 GOSUB 220
>150 PRINT DATE$
>160 GOSUB 220
>170 PRINT "TITLE: ":A$
>180 PRINT "PUBLISHED BY TI"
>190 PRINT PCOUNT;" PAGES"
>200 GOSUB 220
>210 STOP
>220 REM SUBROUTINE TO PRINT
 FILLERS ON A PAGE
>230 PRINT
>240 PRINT "* * * * * * * * *
  * * * * *"
>250 PRINT " * * * * * * * *
 * * * * *"
>260 PRINT
>270 RETURN
```

```
>NEW

>100 REM BUILD AN ARRAY,
 MULTIPLY EACH ELEMENT BY 3,
 PRINT BOTH ARRAYS
>110 FOR X=1 TO 4
>120 FOR Y=1 TO 7
>130 I(X,Y)=INT(30*RND)+1
>140 NEXT Y
>150 NEXT X
>160 PRINT "FIRST ARRAY":
>170 GOSUB 260
>180 FOR X=1 TO 4
>190 FOR Y=1 TO 7
>200 I(X,Y)=3*I(X,Y)
>210 NEXT Y
>220 NEXT X
>230 PRINT "3 TIMES VALUES IN
 FIRST ARRAY"::
>240 GOSUB 260
>250 STOP
>260 REM SUBROUTINE TO PRINT
 ARRAY
>270 FOR X=1 TO 4
>280 FOR Y=1 TO 7
>290 PRINT I(X,Y);
>300 NEXT Y
>310 PRINT
>320 NEXT X
>330 PRINT
>340 RETURN
>RUN
 FIRST ARRAY

  16  12  17  12  8  17  8
  18  22  1  29  16  14  11
  5  25  22  4  24  11  24
  26  21  18  2  12  20  15

 3 TIMES VALUES IN FIRST ARRA
 Y

  48  36  51  36  24  51  24
  54  66  3  87  48  22  33
  15  75  66  12  72  33  72
  78  63  54  6  36  60  45

 ** DONE **
```

Debugging Your Programs

You may find that when you enter and run program, it does not work as you intended. The problems you have in your program are called logical errors or, in computer usage, "bugs." Testing your program to discover these logical errors or bugs is called debugging a program.

When you have bugs in a program, remember that the computer does exactly what you tell it to do. Think about the things that could be causing the errors and try to devise tests to find these errors.

One technique that you can use is to follow the flow of your program, line by line, and on a sheet of paper, write the values of variables at each line. Another technique is to insert lines at various places in your program to print the values of variables.

Debugging Aids

TI-99/2 BASIC has a number of aids that you can use to help you find program errors.

The BREAK Key

The BREAK key can be used to stop program execution and allow you to print or change the values of variables. When the BREAK key is pressed, the program halts program execution and the message BREAKPOINT AT line-number is displayed. The BREAK key is especially useful when your program gets in an infinite loop.

The BREAK Statement

The BREAK statement can be used to cause your program to halt at specific points in your program, in order to enable you to print or change the values of variables.

```
   .
   .
   .
   230 N=A+C*3
   240 IF(N<5) THEN 270
   .
   .
   .
```

In the program section above, you can cause the program to halt before and after the IF statement is performed by using the BREAK statement. The following program sections show two ways of using the BREAK statement.

```
                                  >5 BREAK 240,250
    .                             .
    .                             .
    .                             .
    >230 N=A+C*3                  >230 N=A+C*3
    >235 BREAK
    >240 IF(N<5) THEN 270         >240 IF(N<5) THEN 270
    >245 BREAK
    >250 X=2.5+C                  >250 X=2.5+C
    .                             .
    .                             .
    .                             .
```

The section on the left has two BREAK statements inserted in the program at lines 235 and 245. When lines 235 and 245 are encountered, the program halts and displays a message informing you at what line the break occurred. You can print the values of variables or change their values during this break.

The section on the right has one BREAK statement inserted at the first of the program. This BREAK statement does not cause a break to occur at line 5. It sets breakpoints immediately before the lines that are specified in the BREAK statement. In this case, line 5 sets breakpoints before lines 240 and 250. Therefore, a break occurs before lines 240 and 250 are executed and you can print the values of variables or change their values at that time.

You can also use the BREAK statement at the same places described above by entering the BREAK statement in the Immediate Mode, before you RUN the program or when you are in the middle of a break. Using BREAK in the Immediate Mode is shown below.

```
    BREAK 240,250
    RUN
```

The CONTINUE Command

After you have checked the values of variables during a breakpoint, you can resume program execution by entering the CONTINUE (or CON) command. Note that you cannot enter the CONTINUE command after editing the program (adding, deleting, or changing program lines).

The UNBREAK Command

The UNBREAK statement is used to remove breakpoints that you have set in your program. Note that the breakpoints you can remove with UNBREAK are the ones that are set as shown in the section on the right of the previous example. The breaks that are set in the left section must be deleted from the program.

The TRACE Command

The TRACE command is used to see the order in which your program lines are executed. After the TRACE command is entered, the line number of each program line is displayed before the statement is performed. TRACE may be entered as a program line or in Immediate Mode. In the example below, the TRACE displays the line numbers 230 and 240. If the condition in line 240 is true, the next line number displayed is 270. Otherwise TRACE displays 250.

```
     .
     .
     .
    >225 TRACE
    >230 N=A+C*3
    >240 IF(N<5) THEN 270
     .
     .
XX BETTER EXAMPLE NEEDED
     .
```

TRACE is especially useful in finding infinite loops. The TRACE command is cancelled when a NEW command or the UNTRACE statement is executed.

The UNTRACE Command

UNTRACE is used to cancel a TRACE that is being performed. UNTRACE may be used in a program line or in immediate mode.

Using Subprograms--Graphics

XX Update for real character set--add NEW command

TI-99/2 BASIC has the standard ASCII character set and a group of special graphics characters that are listed in Appendix XX (Book 4). The following program uses the special graphics that are defined for your computer. The program is separated into small parts, which are described in order to help you learn how to use the special graphics in a program.

This program places a square in the middle of the screen and then starts a rocket up the left side of the screen, turns it to the right at the top of the screen, and begins firing arrows at the square. When the arrows hit the square, blocks of the square begin falling off until the block disappears. The rocket then falls to the ground and explodes. All of the special graphics characters that are available on the TI-99/2 are then displayed.

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

The following statement is used to define a variable that determines the length of time a symbol is displayed before it is erased. PAUS is used by the subroutine at line 1100 to determine how many times the subroutine performs a loop.

```
>170 PAUS=30
```

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

Before any characters are displayed, the screen is cleared.

```
>180 CALL CLEAR
```

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

The large square (code 97) graphics symbol is displayed in the middle of the screen.

```
>190 CALL HCHAR(12,14,97)
```

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

The loop in the following section displays the rocket ship (code 20) starting at the 20th row and the 3rd column. The GOSUB 1100 transfers to a subroutine in the program that is used to keep the graphics symbol displayed for a specific length of time before it is erased. PAUS is used in the subroutine to determine how many times the subroutine performs the loop; in this case PAUS has a value of 30. When the subroutine finishes, the symbol is erased when the space character (code 32) is displayed at the location of the rocket.

```
>200 FOR ROW=20 TO 3 STEP -1
>210 CALL HCHAR(ROW,3,20,1)
>220 GOSUB 1100
>230 CALL HCHAR(ROW,3,32,1)
>240 NEXT ROW
```

****
****
****

When the rocket reaches the top of the screen, the rocket that is pointed to the right (code 21) is displayed. The loop moves the symbol from column to column across the 3rd row. In each column the rocket is displayed while the pause subroutine goes through its loop. The rocket is then erased in each column by displaying the space character in its location.

```
>250 FOR COL=3 TO 14
>260 CALL HCHAR(3,COL,21,1)
>270 GOSUB 1100
>280 CALL HCHAR(3,COL,32,1)
>290 NEXT COL
```

****
****
****

When the rocket is over the square, the rocket that is pointed downward (code 19) is displayed. Note that the pause variable used by the subroutine is three times as long as before.

```
>300 CALL HCHAR(3,14,19,1)
>310 PAUS=90
>320 GOSUB 1100
```

****
****
****

FIRE is a subroutine that displays the downward arrow (code 26) moving toward the square.

```
>330 GOSUB 1200
```

****
****
****

When the square is hit by the arrows, the square breaks apart and a portion of the square (code 99) is displayed in the middle of the screen.

```
>340 CALL HCHAR(12,14,99,1)
```

****
****
****

This next section of code displays the fragments of the box (code 100) falling down the screen. Each fragment is displayed while the pause subroutine goes through its loop 30 times. The fragment is then erased with the space character and displayed at a new location.

```
>350 CALL HCHAR(12,13,100,1)
>360 PAUS=30
>370 GOSUB 1100
>380 CALL HCHAR(12,13,32,1)
>390 CALL HCHAR(12,12,100,1)
>400 GOSUB 1100
>410 CALL HCHAR(12,12,,32,1)
>420 CALL HCHAR(13,11,100,1)
>430 GOSUB 1100
>440 CALL HCHAR(13,11,32,1)
>450 GOSUB 1100
```

****
****
****

When the fragments have disappeared, the rocket begins firing again.

```
>460 GOSUB 1200
```

****
****
****

As the arrows hit, the portion of the original square is replaced with a smaller portion (code 107). Again small blocks (code 98) fall from the square. The blocks are displayed while the subroutine loops 30 times and are then erased.

```
>470 CALL HCHAR(12,14,107,1)
>480 CALL HCHAR(13,15,98,1)
>490 GOSUB 1100
>500 CALL HCHAR(13,15,32,1)
>510 CALL HCHAR(14,16,98,1)
>520 GOSUB 1100
>530 CALL HCHAR(14,16,32,1)
>540 CALL HCHAR(16,17,98,1)
>550 GOSUB 1100
>560 CALL HCHAR(16,17,32,1)
>570 GOSUB 1100
```

****
****
****

After the blocks have disappeared, the rocket fires again.

```
>580 GOSUB 1200
```

****
****
****

When the arrows hit, the square is erased and two pieces (code 98) of the square are displayed splitting apart and falling.

```
>590 CALL HCHAR(12,14,32,1)
>600 CALL HCHAR(12,15,98,1)
>610 CALL HCHAR(13,13,98,1)
>620 GOSUB 1100
>630 CALL HCHAR(12,15,32,1)
>640 PAUS=10
>650 GOSUB 1100
>660 CALL HCHAR(13,13,32,1)
>670 CALL HCHAR(14,12,98,1)
>680 CALL HCHAR(13,16,98,1)
>690 PAUS=30
>700 GOSUB 1100
>710 CALL HCHAR(14,12,32,1)
>720 PAUS=10
>730 GOSUB 1100
>740 CALL HCHAR(13,16,32,1)
>750 CALL HCHAR(16,11,98,1)
>760 CALL HCHAR(14,18,98,1)
>770 PAUS=30
>780 GOSUB 1100
>790 CALL HCHAR(16,11,32,2)
>800 PAUS=10
>810 GOSUB 1100
>820 CALL HCHAR(14,18,32,2)
>830 PAUS=30
>840 GOSUB 1100
>850 GOSUB 1100
>860 CALL HCHAR(3,14,32,1)
```

****
****
****

After the square disappears, the rocket falls to the bottom of the screen. The rocket pointed to the right (code 21) is displayed in the 14th column of each row, starting at row 1.

```
>870 FOR ROW=1 TO 20
>880 CALL HCHAR(3+ROW,14,21,1)
>890 GOSUB 1100
>900 CALL HCHAR(3+ROW,14,32,1)
>910 NEXT ROW
>920 PAUS=6
>930 GOSUB 1100
```

****
****
****

After the rocket reaches the bottom of the screen, all of the special graphics characters are displayed in three slanted lines from the bottom of the screen.

```
>940 FOR ROW=1 TO 9
```

****
****
****

The following section displays graphics characters 0 through 8.

```
>950 CALL HCHAR(20-ROW,14-ROW,-1+ROW,1)
>960 PAUS=60
>970 GOSUB 1100
```

****
****
****

The following section displays graphics characters 9 through 17.

```
>980 CALL HCHAR(20-ROW,14,8+ROW,1)
>990 GOSUB 1100
```

****
****
****

The following section displays characters 18 through 27.

```
>1000 CALL HCHAR(20-ROW,14+ROW,17+ROW,1)
>1010 GOSUB 1100
>1020 NEXT ROW
```

****
****
****

This section displays graphics characters 96 through 111 at the bottom of the screen.

```
>1030 FOR COL=1 TO 16
>1040 CALL HCHAR(22,5+COL,95+COL,1)
>1050 NEXT COL
>1060 STOP
```

****
****
****

This subroutine is used to display a graphics character on the screen long enough for it to be seen. You decide how many times the subroutine performs a loop by defining the variable PAUS before the GOSUB to line 1100 is performed. The variable PAUS in effect determines how long a character is displayed before it is erased.

```
>1100 REM SUBROUTINE TO SIMULATE PAUSE
>1110 FOR Y=1 TO PAUS
>1120 NEXT Y
>1130 RETURN
```

****
****
****

This subroutine fires the arrows at the square.

```
>1200 REM THIS SUBROUTINE FIRES ARROWS DOWNWARD
```

Three arrows are displayed in three rows.

```
>1210 PAUS=25
>1220 FOR ROW=4 TO 6
>1230 CALL HCHAR(ROW,14,26,1)
>1240 GOSUB 1100
>1250 NEXT ROW
```

The arrow is erased in row 4 and an arrow is added in row 7. The pause subroutine causes a time delay between the erasure and the display to simulate movement. The loop continues adding and erasing arrows until arrows are displayed in rows 9, 10, and 11.

```
>1260 FOR ROW=4 TO 8
>1270 CALL HCHAR(ROW,14,32,1)
>1280 GOSUB 1100
>1290 CALL HCHAR(ROW+3,14,26,1)
>1300 GOSUB 1100
>1310 NEXT ROW
```

The arrows in 9, 10, and 11 are erased in the loop below.

```
>1320 FOR ROW=9 TO 11
>1330 CALL HCHAR(ROW,14,32,1)
>1340 GOSUB 1100
>1350 NEXT ROW
>1360 PAUS=30
>1370 RETURN
```

File Processing

Your Basic Computer 99/2 is equipped with a powerful programming tool: the capability of storing both programs and data on peripheral devices. Stored programs can be loaded into memory and run. Data can be stored and programs created to update these data. TI-99/2 BASIC provides an extensive range of file-processing features.

A collection of data sent to a peripheral device is called a file. A program may be saved on a memory storage device as a file, and it is sent to a printer as a file. A set of data that can be read, stored, and updated is also called a file. File processing is a term that refers to printing data, saving programs, loading saved programs, deleting saved programs, and reading, storing, and updating data.

Listing a Program to a Printer

You have already learned how to list the lines of a program in memory to the screen by using the LIST command. The LIST command can also be used to print the lines of a program to a printer.*

LIST "HEXBUS.10"	lists the program in memory to device #10 (the Printer/Plotter).
LIST "HEXBUS.20"	lists the program in memory to device #20 (which is attached to the RS232 serial port).
LIST "HEXBUS.50"	lists the program in memory to device #50 (which is attached to the RS232 parallel output port).
LIST "HEXBUS.10",80-150	lists lines 80 through 150 to the Printer/Plotter.
LIST "HEXBUS.20.B=4800,N=10",200-	lists lines 200 through the end of the program to device #20. Software options required to match the device characteristics are specified.

Refer to the RS232 and Printer/Plotter peripheral manuals and other appropriate peripheral manuals for information on software options.

*NOTE: The word HEXBUS is not hyphenated in a command or statement input to the computer.

Saving a Program

The SAVE command is used to write a copy of the program in memory to a peripheral device such as the Wafertape™ Drive or an audio cassette recorder. You can load this saved copy by using the OLD command.

Saving a Program on a HEX-BUS™ peripheral

If you are using a new storage medium such as a Wafertape™ cartridge, you must initialize it before you can write programs and data to it. Initializing storage media is discussed later under "Initializing a Storage Medium on a HEX-BUS™ Peripheral."

The format for the SAVE command is

SAVE HEXBUS.<u>device-number</u>.<u>filename</u>

To SAVE the program in memory named MYPROG to the cartridge in device 1, the SAVE is entered as shown below.

```
SAVE HEXBUS.1.MYPROG
```

The program is written to and stored in the cartridge in device 1 with the filename MYPROG. When you save a program on a tape that contains other programs, be sure to assign to the program in memory a filename that does not already exist on the tape. Otherwise, the program on the tape is erased.

Saving a Program on a Cassette

The SAVE command is also used to save a program to a TI Program Recorder or a compatible recorder cassette recorder. The form for this command is

```
SAVE CS1
```

When the SAVE CS1 command is entered, the computer prints instructions on the screen to tell you how to use the recorder. Follow the directions as they appear on the screen.

After the program has been copied, the computer asks if you want to check the tape to be sure your program was recorded correctly. If you press <u>N</u>, the flashing cursor appears at the left of the screen. You can then type any BASIC commmand. If you press <u>Y</u>, directions for activating the recorder appear on the screen.

If an error occurred, you may choose one of these three options:

Press <u>R</u> to record your program again. The same instructions listed previously reappear.

Press <u>C</u> to repeat the checking procedures. At this point, you may wish to adjust the recorder volume and/or tone controls.

Press <u>E</u> to exit from the recording procedure. The computer directs you to stop the cassette and press <u>ENTER</u>. An error message appears on the screen to inform you that the SAVE command did not properly record your program. After checking your recorder, you can try to record the program again. When the flashing cursor reappears on the screen, enter any BASIC command.

When the SAVE command is performed, the program remains in memory, whether or not an error occurs in recording.

Refer to the SAVE command in Book 4 for further information on saving programs to peripheral devices.

Loading a Stored Program

The OLD command is used to load a program from a peripheral device into memory. The program can then be run with the RUN command. When an OLD command is executed, any open files are closed and any program in memory is automatically cleared before the next program is loaded.

Loading a Program Stored on a HEX-BUS peripheral

The format of the OLD command used to load programs stored on Wafertape™ cartridges is

```
OLD HEXBUS.device-number.filename
```

where device-number is the number of the peripheral device where the program is stored and filename is the name of the file used in the SAVE command to save the program.

```
OLD HEXBUS.1.MYPROG
```

The example above loads a copy of the program in the file MYPROG on the cartridge in device 1 to computer memory.

Loading a Program Stored on a Cassette

The format for loading a program from a cassette with the OLD command is

```
OLD CS1
```

When the OLD CS1 command is entered, the computer prints instructions on the screen for you to follow.

If the computer did not successfully read your program into memory, an error occurs and you may choose either of these options:

Press R to repeat the reading procedure. Before repeating the procedure, be sure to check the items listed on page XX.

Press E to exit from the reading procedure. An error message is displayed, indicating that the computer did not properly read your program into memory.

When the flashing cursor reappears on the screen, you can enter any BASIC command.

Refer to the OLD command in Book 4 for further information on loading previously saved programs from peripheral devices.

Using_Files_to_Read,_Store,_and_Update_Data

You can store, update, and print data to an external device by using the BASIC statements INPUT, PRINT, and RESTORE. However, before you use these statements you must always open a file on a peripheral device with the OPEN statement. The OPEN statement informs the computer how the data on the file are stored and the number that will be used to access the file. You do not use the OPEN statement when you use the BASIC commands SAVE, OLD, or LIST.

When data are read, stored, and updated in a file, you must specify how the data are stored. The following sections describe the attributes among which you can choose as you structure your data files.

Data Format

When data are stored, updated, or printed, you must specify to the computer the data format or how the data are to be recorded. When data are to be printed for people to read, they should be recorded in ASCII characters (like the characters displayed on the screen); this type of data format is called DISPLAY. When DISPLAY data are printed, the numeric and string items are written according to the specificatons in PRINT and appear the same as if the items were displayed on the screen.

When data are stored on a mass-storage device, they should be recorded in INTERNAL (machine language) format. Data written in INTERNAL format is stored in binary code, the type the computer uses to process data. Storing data in this format expedites processing and reduces the storage space required because the computer does not have to convert INTERNAL format to DISPLAY format and back again. When INTERNAL data are used, the numeric and string items are stored as shown below.

!o! Numeric items are stored in a form that occupies 9 bytes of memory. The first byte is used to store the length of the numeric data item (which is always 8) and the 8 bytes are used to store the data value.

(DIAGRAM FP #2)

!o! String data are stored in the same manner, except that the maximum length for a string item extends to 256 bytes. The first byte is used to store the length of the string data and the remaining 0 through 255 bytes are used to store the string value.

(DIAGRAM FP#3)

Data Records

Data are stored, updated, and printed in a form called a record. A record consists of one or more of the processing units called fields and a collection of records is called a file. Records are numbered from 0 through 32767 where record #0 is the first record of the file, record #1 is the second record of the file, and so on.

File organization

When you store and update files on mass-storage devices, the records can be arranged in sequence or in random order. If you want data to be stored so that they may be read in sequence from the beginning, the file should be organized sequentially. Data stored in a sequential file are read in the same way as they are read in a DATA statement. Files kept on tape must be sequential files. When you use external devices to print data for people to read, the records are always processed in sequence beginning with the first record.

If you want to process data directly without reading through all the data in sequence, the file should be organized as a relative (or random access) file. You must specify that a file is relative when you use the OPEN statement to open the file. With relative files you can access a particular record by using the REC clause in the INPUT, PRINT, and RESTORE statements. (The REC clause does not work with sequential files.) Relative files can also be accessed sequentially. Only certain types of devices support relative files.

Record Type

The record type specifies whether the records on a file are all the same length (FIXED) or vary in length (VARIABLE). The keywords FIXED and VARIABLE can be followed by a numeric expression specifying the maximum length of a record. If the length is omitted, the computer assumes a the maximum record length. The maximum record length of the Wafertape™ peripheral is 255 bytes. When you design your records, be familiar with the lengths of the fields that make up a record. Plan your record so that you allow for the largest length needed.

Relative files use only FIXED-length records. Sequential files can have FIXED- or VARIABLE-length records. If record type is omitted for a sequential file, VARIABLE-length records are assumed.

If records are FIXED, the computer pads each record on the right to ensure that it is the specified length. If the data are recorded in DISPLAY format, the computer pads the record with spaces. If INTERNAL format is used, the FIXED length record is padded with binary zeros.

The record length you specify determines how much space is reserved in computer memory for storing a record of the file. If you attempt to write data longer than the record length you specified, the computer breaks the data into separate records. If you write a record that is smaller than the record length you specified, the record occupies only as much space in the file as is required to write its fields of data. When a record is read from a mass-storage device, the computer determines the length of the record by indicators that were written when the record was created.

BASIC Statements and Commands for Processing Data

The TI-99/2 BASIC statements and commands provided for processing data are OPEN, CLOSE, INPUT, PRINT, and RESTORE. The OPEN statement must be used to provide a link between a file number and a file or device. In setting up this link, the OPEN statement specifies how the file can be accessed (for input and/or output) and how the file is organized. The OPEN statement must be executed in a program before any BASIC statement attempts to use a file or device requiring a file number.

The OPEN Statement for HEX-BUS™ Peripherals

The OPEN statement sets up a link between a peripheral device and a file number to be used in all the BASIC statements that refer to the file. In the OPEN statement you specify file attributes such as file accessibility, file organization, record length, and file type. The computer then creates the file according to the specifications in the OPEN statement. When you use the OPEN statement to open a file that already exists, the file attributes you specify must match those you used when the file was created. Note that file accessibility does not have to match the one used when the file was created.

For each opened file, the computer keeps an internal counter that points to the next record to be accessed. The counter is incremented by 1 each time a record is read or written. For random access files, be sure to use the REC clause if you read and write records on the same file within a program. Because the same internal counter is incremented when records are either read from or written to the same file, you could skip some records and write over others if REC is not used.

The following section describes the attributes that can be specified in the OPEN statement and the default values that are assumed if an attribute is omitted.

File Accessibility: The open-mode attribute of the OPEN statement specifies how the file can be accessed. UPDATE is assumed if no open-mode is specified.

Open-mode Attribute	File Accessibility
INPUT	The computer can only read the file.
OUTPUT	The computer can only write to the file.
UPDATE	The computer can both read from and write to the file.
APPEND	The computer can write data only at the end of the file. The records that already exist on the file cannot be accessed.

File Organization: SEQUENTIAL for sequential files or RELATIVE for random-access files. If file organization is omitted, SEQUENTIAL is assumed.

File Types (Data Formats):	DISPLAY or INTERNAL; if file type is omitted, DISPLAY is assumed.
Record Type:	FIXED or VARIABLE followed by a numeric expression specifying the record length. If record type is omitted, FIXED is assumed for RELATIVE files and VARIABLE is assumed for SEQUENTIAL files. If the numeric expression is omitted, the maximum record length is established by the peripheral device.
File Life:	Files are permanent, not temporary. If file life is omitted, PERMANENT is assumed.

The statement below contains only the required specifications in an OPEN statement.

```
    OPEN #7:"HEXBUS.1.BFILE"
```

The file BFILE on device 1 is referenced with a file #7 and uses all the default options as shown below.

!o! The file can be both read from and written to (UPDATE mode).

!o! The file has SEQUENTIAL organization.

!o! The file is stored in DISPLAY data format.

!o! Because this is a SEQUENTIAL file and the record-type is omitted, VARIABLE length records are assumed. The maximum length of a record depends on what peripheral device is used.

In the example below, the OPEN statement opens a file that is to be referenced as #5 in all of the BASIC statements that access the file. The file is opened on device 100 (which is assumed to support relative files) with the filename AFILE. The attributes of the file are RELATIVE (random-access) file organization, INTERNAL format, INPUT open-mode, and a FIXED record length of 64 bytes.

```
    OPEN #5:"HEXBUS.100.AFILE",RELATIVE,INTERNAL,INPUT,FIXED 64
```

The OPEN Statement for the Cassette Recorder

The following section describes the attributes that can be specified in the OPEN statement for the cassette recorder and the default values that are assumed if an attribute is omitted. The at ibutes that are followed by an asterisk (*) must appear in the OPEN stat nt.

File-number*	any number from 1 through 255.
Filename*	CS1
File organization	SEQUENTIAL
Open-mode*	INPUT or OUTPUT
Record-type*	FIXED

For cassette tape records, you may specify any length up to 192 positions. However, the cassette tape device uses records with 64, 128, or 192 positions and pads the record you specify to the appropriate length. Thus, if you specify an 83-position cassette record, the computer actually writes a 128-position record. If the record length is not specified, a 64-position record length is assumed.

When using a cassette recorder, the computer does not compare the file specifications in the OPEN statement to th characteristics of an existing file.

When the computer performs the OPEN statement for a cassette recorder, instructions are printed on the screen for activating the recorder as shown below.

```
>100 OPEN #2:"CS1",INTERNAL,INPUT,FIXED
>.
>. (Program lines . . .)
>.
>290 CLOSE #2
>300 END

* REWIND CASSETTE TAPE   CS1
  THEN PRESS ENTER

* PRESS CASSETTE PLAY    CS1
  THEN PRESS ENTER
  .
  . (Program runs . . .)
  .
* PRESS CASSETTE STOP    CS1
  THEN PRESS ENTER
```

The CLOSE Statement

The CLOSE statement breaks the link between the file-number and the peripheral device. You cannot access this file until you OPEN it again. If you attempt to close a file that is not open, an error message is displayed. The CLOSE statement can be used to delete a file on some peripheral devices. Refer to the appropriate peripheral manual for more information.

The following statements open the file CFILE on device 2, read three values, and close the file.

```
    >100 OPEN #3:"HEXBUS.2.CFILE",INTERNAL
    >110 INPUT #3: NAME$,HOURS,WAGE
    >120 CLOSE #3
```

Initializing a Storage Medium on a HEX-BUS™ Peripheral

To initialize or format a storage medium on a HEX-BUS™ peripheral such as a wafer in the Wafertape™ peripheral, the OPEN statement with the FORMAT MEDIA option is used as shown below.

```
    OPEN #file-number:"HEXBUS.FORMAT MEDIA.device-code"
```

To save a program on a new cartridge in Wafertape™ device #1, the following statements can be used.

```
    OPEN #5:"HEXBUS.FORMAT MEDIA.1"
    CLOSE #5
    SAVE HEXBUS.1.MYPROG
```

The DELETE Statement

The DELETE statement can be used to delete a file from an external storage device, as well as to delete lines from a program (described earlier in "Editing Program Lines").

```
    DELETE "HEXBUS.1.MYFILE"              (Deletes the file named MYFILE on
                                          device 1.)
```

The INPUT Statement

The INPUT statement is used to read data values from a file. You must use the same file-number to read this file as you did to open the file. When the INPUT statement is executed, the data read from the file are assigned to the variables listed in the INPUT statement.

Filling the INPUT Variable-List

When the computer reads a file, it retrieves and stores an entire record in a temporary storage area called an input/output (I/O) buffer. Values are then assigned to the variables in the variable-list from left to right, using the data items (or fields) in the I/O buffer. A separate buffer is provided for each file opened.

If the variable-list of the INPUT statement is longer than the number of fields held in the I/O buffer, the computer retrieves the next record from the file and uses its fields to complete the variable-list. When a variable-list has been filled with the corresponding values, the fields left in the buffer are discarded unless the INPUT statement ends with a comma (as described later in "Pending Input Conditions").

The computer knows the length of each data item in a DISPLAY data record by the commas placed between items. Each item in a DISPLAY data record is checked to ensure that numeric values are stored in numeric variables. If the data do not match the variable-type, an INPUT ERROR occurs and the program terminates.

The computer knows the length of each INTERNAL item by interpreting the one-position length indicator at the beginning of each item. Limited validation of INTERNAL data items is performed. All numeric items must be 9 positions long and must be valid representations of floating-point numbers. Otherwise, an INPUT ERROR occurs and the program terminates.

For FIXED-length, INTERNAL records, reading beyond the actual data recorded in each record causes binary zeros to be read. If you attempt to assign these characters to a numeric variable, an INPUT ERROR occurs. If strings are being read, a null string is assigned to the string variable.

The statements below open a file that is referred to as #3. The computer reads a record into the input buffer and assigns the fields in the record to the variables in the INPUT statement. If there are more fields in the buffer than are needed to assign to the variables, the remaining fields are discarded. If there are not enough fields to assign, the computer reads another record.

```
    OPEN #3:"HEXBUS.1.MYFILE",INTERNAL
    INPUT #3:NAME$,EMPNUMBER,SOCSEC$
```

Pending Input Conditions

An INPUT statement that ends with a comma creates a pending input condition. Any remaining fields in the input buffer are maintained for the next INPUT statement that reads the file. If this next INPUT statement has no REC clause, the computer starts assigning the remaining fields in the buffer to the variables in the INPUT statement. If the INPUT statement contains a REC clause, the remaining fields are discarded and the specified record is read into the I/O buffer. If a pending input condition exists when a PRINT, RESTORE, or CLOSE statement accesses the file, the remaining fields are also discarded.

The statements below open a file and create a pending input condition. After the variables are assigned in the first INPUT statement, any values left in the input buffer are retained. When the next INPUT statement is executed, the remaining values are assigned to the variables.

```
    OPEN #3:"HEXBUS.1.MYFILE",INTERNAL,VARIABLE 64
    INPUT #3:NAME$,EMPNUMBER,SOCSEC$,
    INPUT #3:ADDR$,AGE
```

When the INPUT statement reads DISPLAY data, the fields are separated by commas. DISPLAY records look like the data in a DATA statement. Therefore, numeric and string items must appear as they do in a DATA statement along with their separators. Each field in a DISPLAY record is checked to ensure that numeric values are placed in numeric variables.

The program below computes the average test score. The student's name is read followed by his test grades.

```
    (Record #0 on file #1) Jones,95,98,65,32,78
    (Record #1 on file #1) Smith,67,87,66,90
    (Record #2 on file #1) Lee,89,88,90,67,90

    >100 OPEN #1:"HEXBUS.1.MYFILE",INTERNAL,VARIABLE 64
    >110 INPUT #1:NAME$,G1,G2,G3,G4,G5
    >120 PRINT NAME$;"AVERAGE IS ";(G1+G2+G3+G4+G5)/5
    >130 GOTO 110
```

The first time the INPUT statement is executed, it assigns the values in the first record to the variables. The average is computed and printed. The second time the INPUT statement is executed, the values in the second record are assigned to the variables. Note, however, that there is no value in the buffer for the last variable. Therefore, the INPUT statement reads the next record and attempts to assign a value to that last variable. An error occurs because the variable is a numeric variable and the value is not a numeric value.

When the INPUT statement reads INTERNAL data, the length byte stored with each data item separates the fields. The only validation performed on INTERNAL data is to ensure that numeric data are 8 characters long.

The PRINT Statement

The PRINT statement writes data values to a file. You must use the same file-number to write to the file as you did when you opened the file. When the PRINT statement is executed, the values of the items in the print-list are written to the file.

To write a record to the end of a sequential file, you can use the open-mode APPEND (but you cannot access the other records in the file). To write to the end of the file in UPDATE mode, you must first read to the end of the sequential file before you write the new record. Using the PRINT statement before the end-of-file is reached results in a loss of data because the PRINT statement always defines a new end-of-file each time it is executed.

The values of the variables in the PRINT statement are written in a temporary storage area called an I/O buffer. A separate buffer is provided for each open file number.

Refer to PRINT (with files) for information on how the PRINT statement writes a record in INTERNAL or DISPLAY data format.

Note that if you print a file in DISPLAY format that the computer later reads, the file must look the same as it does in a DATA statement. You must include the comma separators and quotation marks needed by the INPUT statement. When the data are read from the file, the computer separates the fields by means of the comma separators placed between them.

```
>100 OPEN #6:"HEXBUS.1.PENDING",DISPLAY,OUTPUT
>110 INPUT NAME$
>120 IF NAME$="$END" THEN 160
>130 INPUT D,E
>140 PRINT #6:NAME$; ","; G1; ","; G2; "," ; G3; ","; G4; "," ; G5
>150 GOTO 110
>160 CLOSE #6
```

If the file in the example above had been opened with a DISPLAY data format, the PRINT to the file must write print separators between the values for them to be read later.

Pending Print Conditions

When a PRINT statement ends with a comma or semicolon, the values of the print-list are retained in the I/O buffer for the next PRINT statement that writes to the file. If this next PRINT statement has no REC clause, the computer places the values of the print-list into the I/O buffer immediately following the fields already there. If the PRINT statement has a REC clause, the computer writes the pending print that is in the I/O buffer to the file at the position indicated by the internal counter. Then the new PRINT statement is executed.

If a pending print condition exists and an INPUT statement that accesses the file is encountered, the pending print record is written to the file at the position indicated by the internal counter and the internal counter is incremented. Then the INPUT statement is performed as usual. If a pending print exists when a CLOSE or RESTORE statement accesses the file, the pending print is executed before the file is closed or restored.

For example, the following statements open a file for output, accept data from the keyboard, and write it to the file until a $END is entered.

```
>100 OPEN #6:"HEXBUS.1.PENDING",INTERNAL,OUTPUT
>110 INPUT NAME$
>120 IF NAME$="$END" THEN 160
>130 INPUT G1,G2,G3,G4,G5
>140 PRINT #6:NAME$,G1,G2,G3,G4,G5,
>150 GOTO 110
>160 CLOSE #6
```

The EOF Function

The EOF function determines if an end-of-file has been reached. The EOF function can be placed before an INPUT statement to test the file status before attempting to read from the file. The value that is returned by EOF is 0 if you are not at the end of the file and -1 if you are at the end of the file.

For example, the statements below open a file and check whether the end-of-file has been reached before trying to read a record. When the end-of-file is reached, the file is closed.

```
>100 OPEN #6:"HEXBUS.2.CFILE",INTERNAL
>115 PRINT "NAME:";"GRADES:"
>120 IF EOF(6) THEN 160
>130 INPUT #6: NAME$,G1,G2,G3,G4,G5
>140 PRINT NAME$;G1;G2;G3;G4;G5
>150 GOTO 120
>160 CLOSE #6
```

The RESTORE Statement

The RESTORE statement can be used to reposition an open file at record zero (for a sequential file) or at a specific record (for a relative file). If RESTORE refers to a relative file and the REC clause is not used, the file is repositioned to record zero.

For example, the statements below open a file referred to as #1, accept data from the keyboard, and write the processed data to the file. The file is then repositioned to the first record. A printer is opened with a file number of 3. The data are read from file #1 and printed on file #3. When the end-of-file is reached, the message END OF DATA is displayed.

```
>100 OPEN #1:"HEXBUS.1.RESFILE",INTERNAL
>110 INPUT "ENTER NAME: ":NAME$
>120 IF A$="END$" THEN 160
>130 INPUT "ENTER GRADES: ":G1,G2,G3,G4,G5
>140 PRINT #1:NAME$,G1,G2,G3,G4,G5,
>150 GOTO 110
>160 RESTORE #1
>170 OPEN #3:"HEXBUS.50",OUTPUT
>180 IF EOF(1) THEN 220
>190 INPUT #1:NAME$,ST$,ZIP$
>200 PRINT #3:NAME$,ST$,ZIP$
>210 GOTO 180
>220 PRINT "END OF DATA"
>230 CLOSE #1
```

Answers to Review on page 25

1.
```
    >100 READ A$,B$,C$,D
    >110 PRINT A$:B$:C$;D
    >120 DATA "HI.","MY NAME IS MARY.","AGE" 10
```

2.
```
    >110 READ X

    >140 DATA 2,4,6,8,10
```

3.
```
    >100 X=90*0.01745329251994
    >110 PRINT SIN(X):COS(X):TAN(
     X)
```

4. A space is printed for an input of 32. Each time the program is run, the computer displays the character matching the character code that you input.

5.
```
    >100 FOR J=1 TO 5
    >110 READ X
    >120 PRINT CHR$(X)
    >130 NEXT J
    >140 DATA 43,47,61,58,59
```

6. a) The _LEN_ function returns the number of characters in the string specified by the argument.

   b) _DATA_ statements can appear anywhere in a program.

   c) X^(1/2) is equivalent to SQR(X).

   d) _EXP_ is the inverse of the LOG function.

   e) ATN(X) calculates the angle whose tangent is X.

   f) The STR$ or string-number function converts the number specified by the argument into a _string constant_.

   g) The _READ_ statement tells the computer to look at values in the DATA statement.

   h) _DATA_ statements alone do not cause the computer to do anything, because they merely store values to be read by the computer.

   i) Three statements assign values to variables in a BASIC program; they are the _LET (assignment)_ statement, the _INPUT_ statement, and the _READ_ and _DATA_ statements.

Answers to Review on page 36

1. Providing a separate variable name for large numbers of variables can make a program large and unmanageable. Arrays allow you to use a single variable name.

2. Subscript

3. Array name
   Subscript

4. Constant
   Numeric Variable
   Numeric Expression

5. Three

6. Element

7. The array A stores numeric values.
   The array A$ stores string values.

8. The element in the second row and the third column.

9. The DIM statement.

10. No, the largest subscript you could write is 15.

11.
```
100 REM USING AN ARRAY
110 DIM DGIT(5)
120 DGIT(1)="ONE"
130 DGIT(2)="TWO"
140 DGIT(3)="THREE"
150 DGIT(4)="FOUR"
160 DGIT(5)="FIVE"
170 INPUT "ENTER DIGIT, 1-5: ":INDEX
180 PRINT INDEX;"IS SPELLED";DGIT(INDEX)
```

Applications Programs

Information Management

Checkbook Balance

```
>100 DIM CNUM(10),CAMT(10),DA
 MT(10)
>110 CALL CLEAR
>120 INPUT "BANK BALANCE? ":B
 ALANCE
>130 PRINT "ENTER EACH OUTSTA
 NDING"
>140 PRINT "CHECK NUMBER AND
 AMOUNT"
>150 PRINT
>160 PRINT "ENTER A ZERO FOR
 THE"
>170 PRINT "CHECK NUMBER WHEN
 FINISHED."
>180 PRINT
>190 N=N+1
>200 INPUT "CHECK NUMBER? ":C
 NUM(N)
>210 IF CNUM(N)=0 THEN 250
>220 INPUT "CHECK AMOUNT? ":C
 AMT(N)
>230 CTOTAL=CTOTAL+CAMT(N)
>240 GOTO 190
>250 PRINT "ENTER EACH OUTSTA
 NDING"
>260 PRINT "DEPOSIT AMOUNT."
>270 PRINT
>280 PRINT "ENTER A ZERO AMOU
 NT"
>290 PRINT "WHEN FINISHED."
>300 PRINT
>310 M=M+1
>320 INPUT "DEPOSIT AMOUNT? "
 :DAMT(M)
>330 IF DAMT(M)=0 THEN 360
>340 DTOTAL=DTOTAL+DAMT(M)
>350 GOTO 310
>360 NBAL=BALANCE-CTOTAL+DTOT
 AL
>370 PRINT "NEW BALANCE= ";NB
 AL
>380 INPUT "CHECKBOOK BALANCE
 ? ":CBAL
>390 PRINT "CORRECTION= ";NBA
 L-CBAL
```

Future Value of Investment

```
>100 REM FUTURE VALUE
>110 INPUT "ENTER PRESENT VAL
 UE: ":PV
>120 INPUT "ENTER % INTEREST:
  ":I
>130 INPUT "ENTER # OF PERIOD
 S: ":N
>140 FV=INT(((PV*(1+I/100)^N)
 +.005)*100)/100
>150 PRINT "FUTURE VALUE= $";
```

Monthly Payment

```
>100 REM MONTHLY PAYMENT
>110 INPUT "ENTER PRESENT VAL
 UE: ":PV
>120 INPUT "ENTER % INTEREST:
  ":I
>130 INPUT "ENTER # OF YEARS:
  ":N
>140 PMT=PV/((1-(1+I/1200)^(-
 N*12))/(I/1200))
>150 PMT=INT((PMT+.005)*100)/
 100
>160 PRINT "PAYMENT= $";STR$(
 PMT)
```

Wind Chill Factor

```
>100 CALL CLEAR
>110 INPUT "ENTER DEGREES F:
 ":TEMP
>120 INPUT "ENTER MPH OF WIND
 : ":VEL
>130 WCF=91.4-(.288*SQR(VEL)+
 .450-.019*VEL)*(91.4-TEMP)
>140 PRINT INT(TEMP);"DEG. &"
 ;INT(VEL);"MPH WIND";"=":INT
 (WCF);"DEG. WIND CHILL FACTO
 R"::
>150 GOTO 110
```

<u>Education</u>

Factorials

```
>100 CALL CLEAR
>110 PRINT "FACTORIALS OF AN
 INTEGER":"FROM 0 TO 69"
>120 PRINT
>130 INPUT "ENTER A NUMBER ":
 N
>140 IF (0<=N)*(N<70) THEN 16
 0
>150 GOTO 170
>160 IF N-INT(N)=0 THEN 190
>170 PRINT "INVALID NUMBER"
>180 STOP
>190 F=1
>200 FOR N=1 TO N
>210 F=F*N
>220 NEXT N
>230 PRINT FACTORIAL OF ";(N-
 1);"IS ";F
```

Math (Multiplication and Division)

```
>100 CALL CLEAR
>110 PRINT "MULTIPLICATION OR
  DIVISION"::
>120 N=0
>130 INPUT "ENTER M FOR MULTI
 PLICATION; D FOR DIVISION ":
 A$
>140 IF A$="M" THEN 170
>150 IF A$="D" THEN 300
>160 GOTO 120
>170 INPUT "FIRST NUMBER ":A
>180 INPUT "TIMES ":B
>190 INPUT "= ":C
>200 IF A*B=C THEN 280
>210 IF N>1 THEN 260
>220 PRINT "WRONG, TRY AGAIN"
 :A;" TIMES ";B;
>230 INPUT " =? ":C
>240 N=N+1
>250 GOTO 200
>260 PRINT "WRONG,":"THE ANSW
 ER IS ":A;" TIMES ";B;" = ";
 A*B
>270 GOTO 450
>280 PRINT "VERY GOOD"
>290 GOTO 450
>300 INPUT "FIRST NUMBER ":A
>310 INPUT "DIVIDED BY ":B
>320 IF 10^6*A/B=INT(10^6*A/B
 ) THEN 350
>330 PRINT "THAT'S TOO DIFFIC
 ULT":"TO CALCULATE,":"TRY AG
 AIN"
>340 GOTO 300
>350 INPUT " = ":C
>360 IF A/B=C THEN 440
>370 IF N>1 THEN 420
>380 PRINT "THAT'S INCORRECT,
  TRY AGAIN": " DIVIDED BY ";
 B;
>390 INPUT " =? ":C
>400 N=N+1
>410 GOTO 360
>420 PRINT "THAT'S INCORRECT.
 ":"THE ANSWER IS ":A;" DIVID
 ED BY ";B;" = ";A/B
>430 GOTO 450
>440 PRINT "EXCELLENT"
>450 INPUT "MORE?(Y/N)":D$
>460 IF D$="Y" THEN 120
>470 IF D$="N" THEN 490
>480 GOTO 450
>490 END
```

Metric Conversion

```
>100 REM UNIT CONVERSIONS
>110 CALL CLEAR
>120 PRINT "UNIT CONVERSIONS"
 : "CHOOSE A NUMBER"
>130 PRINT
>140 PRINT "1. INCHES TO CENT
 IMETERS"
>150 PRINT "2. CENTIMETERS TO
  INCHES"
>160 PRINT "3. FAHRENHEIT TO
 CELSIUS"
>170 PRINT "4. CELSIUS TO FAH
 RENHEIT"
>180 PRINT "5. MILES TO KILOM
 ETERS"
>190 PRINT "6. KILOMETERS TO
 MILES"
>200 PRINT
>210 INPUT "ENTER A NUMBER:":
 N
>220 CALL CLEAR
>230 IF N=1 THEN 300
>240 IF N=2 THEN 350
>250 IF N=3 THEN 400
>260 IF N=4 THEN 450
>270 IF N=5 THEN 500
>280 IF N=6 THEN 550
>290 GOTO 100
>300 INPUT "INCHES? ":I
>310 C=I*2.54
>320 PRINT I;"INCHES=";C;"CEN
 TIMETERS"
>330 GOTO 600
>340 STOP
>350 INPUT "CENTIMETERS? ":C
>360 I=C/2.54
>370 PRINT C;"CENTIMETERS=";I
 ;"INCHES"
>380 GOTO 600
>390 STOP
>400 INPUT "FAHRENHEIT? ":F
>410 C=(F-32)*5/9
>420 PRINT F;"FAHRENHEIT=";C;
 "CELSIUS"
>430 GOTO 600
>440 STOP
>450 INPUT "CELSIUS? ":C
>460 F=C*9/5+32
>470 PRINT C;"CELSIUS=";F;"FA
 HRENHEIT"
>480 GOTO 600
>490 STOP
>500 INPUT "MILES? ":M
>510 K=M*1.609344
>520 PRINT M;"MILES=";K;"KILO
 METERS"
>530 GOTO 600
>540 STOP
>550 INPUT "KILOMETERS? ":K
>560 M=K/1.609344
>570 PRINT K;"KILOMETERS=";M;
 "MILES"
```

```
>590 STOP
>600 PRINT
>610 PRINT "WOULD YOU LIKE TO
  DO SOME":"MORE?"
>620 INPUT "ENTER Y OR N: ":A
 $
>630 CALL CLEAR
>640 IF A$="Y" THEN 120
>650 STOP
```

Entertainment

Blackjack

```
>100 REM BLACKJACK
>110 RANDOMIZE
>120 CALL CLEAR
>130 PRINT "PLAYER ";
>140 A=2
>150 GOSUB 280
>160 A=1
>170 CALL KEY(0,K,S)
>180 IF S=0 THEN 170
>190 IF K=72 THEN 150
>200 PRINT "DEALER ";
>210 A=2
>220 GOSUB 280
>230 A=1
>240 CALL KEY(0,K,S)
>250 IF S=0 THEN 240
>260 IF K=72 THEN 220
>270 GOTO 100
>280 FOR I=1 TO A
>290 N=INT(13*RND)+1
>300 IF N=1 THEN 360
>310 IF N=11 THEN 380
>320 IF N=12 THEN 400
>330 IF N=13 THEN 420
>340 C$=STR$(N)
>350 GOTO 430
>360 C$="A"
>370 GOTO 430
>380 C$="J"
>390 GOTO 430
>400 C$="Q"
>410 GOTO 430
>420 C$="K"
>430 PRINT C$;" ";
>440 NEXT I
>450 RETURN
```

Call Key

This program places a black cursor in the middle of the screen. Using the S, D, E, X keys, you can draw anything you like with the black cursor.

```
>100 CALL CLEAR
>110 R=12
>120 C=16
>130 CALL HCHAR(R,C,30)
>140 CALL KEY(0,K,S)
>150 IF S=0 THEN 140
>160 IF K=69 THEN 210
>170 IF K=68 THEN 260
>180 IF K=83 THEN 310
>190 IF K=88 THEN 360
>200 GOTO 140
>210 R=R-1
>220 IF R<1 THEN 240
>230 GOTO 130
>240 R=24
>250 GOTO 130
>260 C=C+1
>270 IF C>32 THEN 290
>280 GOTO 130
>290 C=1
>300 GOTO 130
>310 C=C-1
>320 IF C<1 THEN 340
>330 GOTO 130
>340 C=32
>350 GOTO 130
>360 R=R+1
>370 IF R>24 THEN 390
>380 GOTO 130
>390 R=1
>400 GOTO 130
```

Simon

```
>100 REM SIMON
>110 CALL CLEAR
>120 PRINT "CHASE ME"
>130 PRINT
>140 INPUT "CHOOSE GAME A OR
 B ":A$
>150 N=0
>160 DIM F(25)
>170 DIM B(25)
>180 RANDOMIZE
>190 N=N+1
>200 IF N=25 THEN 510
>210 F(N)=INT(10*RND)
>220 FOR K=1 TO N
>230 DISPLAY F(K)
>240 FOR J=1 TO 40
>250 NEXT J
>260 CALL CLEAR
>270 NEXT K
>280 IF A$<>"A" THEN 350
>290 FOR M=1 TO N
>300 INPUT B(M)
>310 IF B(M)<>F(M) THEN 430
>320 CALL CLEAR
>330 NEXT M
>340 GOTO 190
>350 IF A$<>"B" THEN 110
>360 PRINT "CHASE ME BACKWARD
 S"
>370 FOR L=N TO 1 STEP -1
>380 INPUT B(L)
>390 IF B(L)<>F(L) THEN 430
>400 CALL CLEAR
>410 NEXT L
>420 GOTO 190
>430 PRINT "OOP!":"YOU SHOULD
  HAVE ENTERED"
>440 FOR I=1 TO N
>450 PRINT F(I)
>460 NEXT I
>470 GOTO 510
>480 FOR I=N TO 1 STEP -1
>490 PRINT F(I)
>500 NEXT I
>510 PRINT "YOUR SCORE IS ":(
 N-1)
>520 PRINT "DO YOU WANT TO TR
 Y AGAIN?"
>530 INPUT "ENTER Y OR N ":B$
>540 IF B$="Y" THEN 120
>550 END
```