


Introduction

The BASIC programming language was developed at Dartmouth College in the 1960s. The word *BASIC* is an acronym for *Beginner's All-purpose Symbolic Instruction Code*. BASIC is the language most commonly used on personal and home computers and is increasingly accepted on larger machines. The Texas Instruments Compact Computer Model CC-40 uses an advanced version of BASIC called CC-40 BASIC. Programs written in other versions of BASIC, including TI BASIC and TI Extended BASIC (used on the Texas Instruments Home Computer), may have to be modified for use on the CC-40.

This chapter provides an introduction to BASIC that enables you to use the BASIC programming features immediately. It contains overviews of what a program is, how to write a program, the rules and syntax of CC-40 BASIC commands, statements, and functions, what input and output are, the various parts of a program, how to edit a program, how to save a program, and how to debug (or find the errors in) a program. Each of the elements that makes up the CC-40 BASIC language is mentioned, and some are explained in detail. For details on syntax and additional examples, see chapter 5, which is an alphabetical listing of all the CC-40 BASIC commands, statements, and functions.

If you have not used BASIC before, the book *Learn BASIC: A Guide to Programming the Texas Instruments Compact Computer* is available from local dealers. The only way to learn to program is to actually program. Try the examples in this book. Don't worry about making mistakes when you begin. You can always cancel any operation by pressing **[BREAK]** and **[CLR]**.

If you are already familiar with some version of BASIC, this chapter is a quick refresher. Be sure to review the topics that are marked with the TI logo () for features of CC-40 BASIC that may differ from other versions of BASIC.

Getting Started

A program is a series of instructions that the computer can perform. You tell the computer what to do by typing instructions on the keyboard and then pressing [ENTER]. These instructions are performed only when you run or execute the program by typing RUN and pressing [ENTER]. When you run a program, the instructions are performed or executed one at a time.

The computer has its own set of words, called keywords, that it knows how to perform. There are not many such words, but taken together these words let you perform virtually any computational task.

As a simple example, you can write a program to multiply 25 times 7 and print the answer. A program to do this is shown in the following section.

Writing, Running, and Listing a Program

Turn on the CC-40. Either a message is displayed or there is a flashing cursor in column 1. To clear the message from the display, press the [CLR] key. Then in both cases, type NEW and press [ENTER].

Type the following line exactly as shown, including spaces. As you type, the cursor moves to the right to show where the next character is placed. If you make a mistake typing, use the edit keys (described in chapter 1) to correct the line, or press the [CLR] key to start over.

```
100 PRINT 25*7
```

Press [ENTER] for the computer to store the instruction in its memory. Type the next line exactly as shown.

```
110 PAUSE
```

Press [ENTER] for this instruction to be stored.

The computer has now stored two instructions or lines in its memory. To have the computer perform these instructions, you must run the program by typing RUN and pressing [ENTER].

The computer calculates 25 times 7 and prints the answer, 175, in the display. It then pauses so you can see the answer. The result is preceded by an underline cursor in column 1. The underline cursor indicates that the computer is waiting for the [CLR] or [ENTER] key to be pressed. To leave the pause, press [CLR]. The flashing cursor appears in column 1 to indicate that you can enter information from the keyboard.

Note: You can leave any BASIC program that is running by pressing [BREAK].

To look at the program in the CC-40, type LIST and press [ENTER]. The CC-40 prints the first line of the program.

```
100 PRINT 25*7
```

Press [ENTER] to display the next line.

```
110 PAUSE
```

Press [ENTER] again and the flashing cursor appears on a blank line. The blank line means there are no more program lines stored in memory. You can now proceed to enter information.

If you are in the middle of a listing and do not want to see the rest of a program, press [BREAK] to end the listing.

BASIC Programming Procedures

The two-line program you just entered, executed, and listed shows many of the procedures to follow as you write BASIC programs. Remember—before you enter a program in the CC-40, type NEW and press [ENTER] to be sure that memory has been cleared.

The rules for entering program instructions are described briefly in the following sections.

Lines and Line Numbering

100 PRINT 25*7 is called a line of a program. This line instructs the computer to calculate 25 times 7 and print the answer. The computer recognizes * as multiplication and the word PRINT as a task to perform.

The 100 is the line number. Every line of a program must have a line number from 1 through 32766, followed by a space. The CC-40 executes program lines in numerical order, regardless of the order in which they are entered.

CHAPTER IV BASIC PROGRAMMING

It is good programming practice to have unused line numbers between the lines of a program. You can then insert additional lines in the program. For example, suppose you want to perform one more calculation in the program, 25 times 6. Enter the following two lines.

```
105 PAUSE  
108 PRINT 25*6
```

If you **LIST** the program, you will find that the two lines have been added to the program in memory. The computer sequences the lines by their line numbers. To run the program, be sure the display is clear and then type **RUN** and press **[ENTER]**. When the computer pauses while displaying the first answer, 175, press **[ENTER]** to display the next answer, 150. Then press **[ENTER]** again.

If you enter a line with the same line number as one already stored in memory, the new line replaces the old one. For example, type the following line and then press **[ENTER]**.

```
100 PRINT 25*25
```

When you list the program, you will find the last line entered as 100 is the one that is now stored in memory.

Note: When entering any information into the computer, always press **[ENTER]** after you have finished typing. In the rest of this chapter, any references to enter any information assume that the **[ENTER]** key is pressed after you have finished typing.

Keywords

Following the line numbers in the program just executed are English words that the CC-40 recognizes and knows how to perform or execute. All program lines contain characters that resemble algebraic formulas and/or English words. These English words correspond to single tasks and are called keywords. When entering keywords in the CC-40, you can type them using either upper- or lower-case characters. When the program is **LISTed**, these words are always displayed in upper-case letters. The different types of keywords are discussed in the following sections.

CHAPTER IV BASIC PROGRAMMING

Statements

Statement keywords are elements of a program line that cause an action, such as **PRINT** and **PAUSE**. Statement keywords must be followed by a space in the program line. For example, the CC-40 recognizes the statement 10 PRINT 2 but not 10 PRINT2. The CC-40 performs statements in a program only when you execute the program.

Many statement keywords can also be executed immediately by entering them without a line number. The statement is executed as soon as the **[ENTER]** key is pressed. For example, enter the following in your CC-40.

```
PRINT 25*7
```

The answer 175 is displayed immediately.

A list of all the statement keywords, indicating which ones can be executed immediately as well as used in program lines, is given in appendix A. For simple calculations such as the one above, it is usually more practical to use the calculator features of your CC-40 (refer to chapter 2). However, when you want to perform the same series of calculations repeatedly, you can often save time and effort by writing these calculations as a program for the CC-40 to perform.

Functions

The function keywords perform specialized routines and return a value. Most functions require that a value (called an argument) be given to the function. There are function keywords for many mathematical functions such as square root, logarithm, and sine.

All of the functions available with the CC-40 are listed in appendix B and many are discussed in this chapter in sections that deal with similar instructions. All of the functions can be used in program lines and most of them can be executed immediately.

Commands

The command keywords, such as **NEW** and **LIST**, are always executed immediately. These keywords may not be used in a program line. The commands available on the CC-40 are listed in appendix A and are discussed throughout this chapter.

Using a Function in a Program

The program below calculates and displays the square roots of the first 25 whole numbers. To obtain these answers without a program would require making 25 separate calculations. Enter the program shown below in your CC-40. (Remember to type **NEW** and press [ENTER] before you enter the program, and to press [ENTER] after you have typed each line.)

```
100 FOR A=1 TO 25
110 PRINT A; SQR(A)
120 PAUSE 1
130 NEXT A
```

In line 100, the FOR statement sets up a loop, a group of statements that are repeated a specific number of times. The loop consists of the statement immediately following the FOR statement and all the statements down to a NEXT statement. In this case, the letter A is the counter that starts at 1 and goes TO 25 by ones.

Line 110 tells the computer to print the value of the counter A and the square root of the counter. The first time line 110 is executed, the value of the counter is 1.

Line 120 tells the computer to pause for 1 second after it prints an answer so that you have time to see it.

Line 130 is the last statement in the loop. The counter, A, is incremented by one and the CC-40 goes back and repeats lines 110, 120, and then 130, where the counter is again incremented. The loop is repeated until the counter is incremented past the number 25, which is the number following the word TO in line 100. Thus, this loop is executed 25 times.

Enter **RUN** to execute the program. The CC-40 displays the first twenty-five whole numbers and their square roots, pausing one second to display each.

Ending a Program

A program normally stops running after the last line in the program has been executed. However, if you wish, you can enter an END statement as the last statement in your program. The STOP statement can be entered anywhere in your program that you want the program to stop execution.

Kinds of Entries

Everything entered in the CC-40 is determined by the CC-40 to be one of two kinds of entries.

1. An entry that begins with a number from 1 through 32766 followed by a space and an alphabetic character (or an at sign, the underscore, or an exclamation point) is treated as a program line that is stored in memory.
2. Any other entry is assumed to be a command, a statement, or a calculation that is executed immediately. Calculations are discussed in chapter 2.

Program Lines

This section describes the requirements and restrictions of program lines including line numbering, line length, lines containing remarks, and multiple statement lines.

Line Numbering

Each line in a program must begin with a number followed by a space. Line numbers can be any integer from 1 through 32766. It is good practice to number lines in multiples of 10 in case you need to insert lines.

Automatic Line Numbering

You can have the CC-40 supply line numbers by entering the command **NUM** (for **NUMBER**). The CC-40 displays 100 followed by a space. The cursor is positioned where the first character of the line starts. After you type the statement and press [ENTER], the CC-40 displays 110 followed by a space and waits for you to enter the statement for that line. When you have finished entering all of the program lines, press either [ENTER] or [BREAK] when the next line number appears.

You can also use NUM to tell the CC-40 where to start numbering and what increment you want. For example, entering **NUM 10,20** starts numbering the lines at 10 and increments each succeeding line number by 20.

Renumbering Program Lines

After editing a program, you may want to renumber the program lines. The CC-40 automatically renumbers the lines in a program when you use the **RENUMBER** (or **REN**) command.

Line Length

A line may be up to 80 characters long, including the line number. Additional characters typed at the end of the line replace the 80th character.

Lines Containing Remarks

You can include explanations and comments in a program by using remarks. A remark is not executed, but it is stored in memory. Enter remarks either by typing **REM** followed by a space and the explanatory remark or by typing an exclamation point and the explanatory remark as shown in lines 100 and 110 below.

```
100 REM THIS LINE IS A REMARK AND IS NOT EXECUTED
110 ! NEITHER IS THIS ONE
```

The exclamation point can also be used as a tail remark symbol by following the statements on a line with the exclamation mark (!) and the explanatory remark, as shown below.

```
120 FOR A=1 TO 25 ! SET UP LOOP
```

Multiple Statement Lines

Each program line may contain more than one statement by separating the statements with colons. For example, the program that calculated the square roots of the first 25 whole numbers could be written on one line as shown below.

```
100 FOR A=1 TO 25:PRINT A;SQR(A):PAUSE 1:NEXT A ! PRINT
    SQUARE ROOTS
```

The line begins with a line number followed by a space. The statement **FOR A=1 TO 25** is followed by a colon to signal the CC-40 that there is another statement, **PRINT A; SQR(A)**. There are four statements in the line. The tail remark symbol (!) tells the CC-40 that the rest of the program line is an explanatory remark which is not to be executed when the program is run.

Program Storage and Execution

You can save a program that you want to keep by using the **SAVE** command. To execute a program that has been stored, use the **RUN** statement or the **OLD** command and **RUN**.

Saving a Program

The **SAVE** command is used to copy a program in memory to an external storage device. To store a program on a new tape, you must first format the tape. If you format a tape that already has information on it, all the data is erased. The example below illustrates how to save a program on a new tape.

```
FORMAT 1
SAVE "1.MYPROG"
```

The tape on device 1 is formatted and the program in memory is written to the tape with the filename **MYPROG**. To save a program on a tape that contains other programs, be sure to give the program in memory a name that does not already exist for a program on the tape.

You can also protect a program when you save it by using **PROTECTED** in the **SAVE** command. If the **SAVE** command includes the option **PROTECTED**, the saved copy can not be listed, edited, or stored. For example, the following **SAVE** command places a protected copy of the program in memory on external device 1.

```
SAVE "1.MYPROG",PROTECTED
```

Note: Since a protected program can never be listed, edited, or stored, be sure to save an unprotected copy.

Executing a Stored Program

To execute a program stored on a peripheral device, the program must be loaded into memory by using the **OLD** command or the **RUN** statement. The **OLD** command is used when you want to edit the program or verify that it was loaded into memory correctly. The commands shown below load a program into memory and verify that it was loaded correctly.

```
OLD "1.MYPROG"
VERIFY "1.MYPROG"
```

To execute the program, enter **[RUN]**.

The **RUN** statement can be used to execute a program stored on a peripheral device. The statement below loads a program into memory from peripheral device 1 and then executes it.

```
RUN "1.MYPROG"
```



Editing Program Lines

After you enter a program, it is often useful to check the program lines for errors by using LIST or the edit keys. Many of the editing features are obtained using the [SHIFT], [CTL], and [FN] keys. By using these keys, you can display lines, delete lines and portions of lines, and move the cursor within a line.

Note: In CC-40 BASIC you cannot delete a line by entering its line number alone. You must use the DELETE keyword described later in this section.

The Right Arrow Key— →

The right arrow key moves the flashing cursor one position to the right. If you press and hold the → key, the cursor continues to move to the right to column 31 and then scrolls the display to the left until column 80 is reached or until the key is released.

The Left Arrow Key— ←

The left arrow key moves the flashing cursor one position to the left. If you press and hold the ← key, the cursor continues to move to the left until it reaches column 1 or until the key is released.

The Up Arrow Key— ↑

The up arrow key is used to display the next lower-numbered program line. If ↑ is pressed with the first line of the program in the display, the CC-40 displays the flashing cursor on a blank line. If you press ↑ again, the highest-numbered program line is displayed. You can also use the ↑ key to display a specific program line by typing the line number and pressing ↑.

The Down Arrow Key— ↓

The down arrow key is used to display the next higher-numbered program line. If ↓ is pressed with the last line of the program in the display, the CC-40 displays the flashing cursor on a blank line. If ↓ is pressed again, the lowest-numbered program line is displayed. You can also use the ↓ key to display a specific program line by typing the line number and pressing ↓.

Character Insert—[SHIFT] [INS]

[SHIFT] [INS] is used to insert characters in a line. The following keys can be used to end an insert.

- leaves the edited line in the display and moves the cursor one position to the right.
- ← leaves the edited line in the display and moves the cursor one position to the left.
- ↑ enters the edited line. If the line was a program line, the next lower-numbered program line is displayed.
- ↓ enters the edited line. If the line was a program line, the next higher-numbered program line is displayed.

[ENTER] enters the edited line. If the line was a program line, the display is cleared. If LIST is in effect, the next higher-numbered program line is displayed.

Character Delete—[SHIFT] [DEL]

[SHIFT] [DEL] is used to remove the character at the position occupied by the flashing cursor. If you press [SHIFT] and then press and hold [DEL], the computer continues to delete characters, one at a time, until [DEL] is released. If you press [SHIFT] [DEL] when you are inserting characters, the insert is ended.

Playback—[SHIFT] [PB]

[SHIFT] [PB] causes the previous display contents to reappear. If you want to enter a line similar to the most recently entered line, press [SHIFT] [PB] and edit the line using the edit keys. The [PB] key can be used to avoid retyping a long line. If you press [SHIFT] [PB] when you are inserting characters, the insert is ended.

Tab—[CTL] →

[CTL] → shifts the display to the next higher-numbered tab position. Tab positions are set at 1, 25, and 50.

Back Tab—[CTL] ←

[CTL] ← shifts the display to the next lower-numbered tab position.

Home—[CTL] ↑

[CTL] ↑ moves the cursor to position 1 of the line.

Erase Field—[CTL] ↓

[CTL] ↓ clears the display from the current cursor position to the end of the line.

Line Delete—DELETE

The DELETE (or DEL) keyword is used to delete a group of program lines. DELETE can be accessed by pressing [FN] [DEL] or by typing DELETE or DEL. You can delete a single line or a group(s) of lines by entering DELETE (or DEL) followed by one or more of the line groups shown below.

Line-group	Effect
a single line number	Deletes that line.
line number —	Deletes that line and all following lines.
— line number	Deletes that line and all preceding lines.
line number — line number	Deletes that inclusive range of lines.

If more than one line number group is used, use commas to separate the groups. For example, **DEL 150,320-350,560-** deletes line 150, lines 320 through 350, and lines 560 through the end of the program.

Error Handling

As you begin to write BASIC programs, you may make mistakes as you enter instructions. The computer tells you through error messages what is wrong. Sometimes a line can not be stored in memory because you have made an error in typing it. Sometimes a program may not work the first time you attempt to execute it. By using the error messages that the computer displays, you can determine what corrections to make.

For example, the following program has two errors in it.

```
100 FOR A = 1 TO25
110 PRINT A; SQR(A
120 PAUSE 1
130 NEXT A
```

When you enter lines 100, 120, and 130, they are stored in memory. However, line 110 causes an error when you try to enter it. The error indicator in the display is turned on and the error message *Unmatched parenthesis* is displayed. To correct this line, press [SHIFT] [PB] to display it, and then add a parenthesis after the last A.

If you try to run the program, the error message *Illegal syntax* is displayed. When an error message is displayed, press → to display the error code and the number of the erroneous line. In this case, the error code that is displayed is E1 and the number of the erroneous line is 100.

Press ↑ or ↓ to display the erroneous line. Between the word TO and the number 25 there must be a space. Use the edit keys to place a space there. You can then run the program.

Refer to appendix K for a list of the error codes and messages. You can handle errors which occur while a program is running by using the error processing statements available in CC-40 BASIC. Refer to *Handling Errors in a BASIC Program* in this chapter.

Constants and Variables

The data used by BASIC keywords may be either constants or variables. The rules and conventions used are described in the following sections.

Constants

A constant is a value that does not change throughout the entire execution of a program. There are two kinds of constants, numeric and string.

Numeric Constants

A numeric constant is either a positive or negative real number or zero. Positive numbers may optionally be written with a + sign. Negative numbers must be preceded by a minus sign. Commas and spaces are not allowed in numbers.

Constants may be entered with any number of digits, but they are rounded to 13 or 14 digits due to the internal storage method used by the CC-40. Only ten digits of a constant are displayed when a program is running, but all 13 or 14 digits are

used in calculations and are displayed when a program is listed.

Numbers are normally stored and displayed in standard notation. Very large or small numbers are stored and displayed in scientific notation, which is described in chapter 2.

For example, if you enter a constant as 3E4, it is retained in memory and displayed when the program is listed or run as 30000.

The following are examples of valid numeric constants.

5
25.7
3.598E4 (which is retained internally as 35980)
-1900

String Constants

A string constant is a series of characters usually enclosed in quotation marks. The quotation marks may be omitted when a string constant is used in a DATA or IMAGE statement. To include leading and trailing blanks in a string constant, you must use quotation marks. A quotation mark within a quoted string constant is represented by two quotation marks. To include a quotation mark at the beginning or end of a string constant, you use three quotation marks. The CC-40 does not change any lower-case alphabetic characters to upper-case characters in string constants.

The following are examples of valid string constants and the way they would appear if printed.

<u>String Constant Example</u>	<u>Appears In Print</u>
Hello""Goodbye	Hello""Goodbye
""""Hello""Goodbye""	"Hello"Goodbye"
Hello Goodbye	Hello Goodbye
" Hello Goodbye"	Hello Goodbye

Variables

A variable is a name given to a memory location in the CC-40. You can store a value in that location, and later change the value of the variable by storing a different value in the location.

A variable name can consist of up to 15 characters, the first of which must be a letter of the alphabet, an underline (), or the

at sign (@). The remaining 14 characters can be alphanumeric, the underline, or @. A program can include up to 95 variable names. The keywords that are reserved for use by CC-40 BASIC may not be used as variable names, but they may make up part of a variable name. See appendix C for a complete list of the words reserved for CC-40 BASIC.

There are two kinds of variables, numeric and string.

Numeric Variables

A numeric variable is a name given to a location that stores a numeric value. The following are valid numeric variable names.

X, A9, @ALPHA, BASE__PAY, __@TABLE4

String Variables

A string variable is a name given to a location that stores any combination of characters (letters, numbers, and other symbols). The string variable name must end with a \$, which is counted as one of the 15 characters allowed. The following are examples of valid string variable names.

N\$, YZ2\$, NAME@\$, Q__505\$, ADDRESS\$

Assigning Values to Variables

Before values are assigned, numeric variables are equal to zero and string variables are assumed to have no characters (or be null). You can set the value of a variable to be either a constant or the result of a calculation by using an assignment statement. You can also set the value of a variable with READ and DATA statements or by various input statements.

Assigning values using READ and DATA statements and input statements is discussed later in this chapter.

In the example below, a 5 is put or stored in the location called K when line 210 is executed and the characters File- are stored in the location called K\$ when line 220 is executed. When line 230 is executed, the value of K*10 (50) is stored in locations A, B, and C.

```
210 K = 5
220 K$ = "File-"
230 A,B,C = K*10
```


CHAPTER IV BASIC PROGRAMMING

You can also use the optional keyword LET in an assignment statement. The following statement stores the result of 25.5 times 3 in the location called A when the line is executed.

```
250 LET A = 25.5*3
```

Arrays

An array is a group of values given the same variable name. Each value is an element of the array. The elements are in an ordered sequence to provide easy access to any value in the array. When a variable name is chosen for an array, that name must always refer to the array. For example, if A is chosen as the name for an array, then A cannot appear as a simple variable elsewhere in the program.

To tell the computer which element you are using, you need a pointer. The pointer, called a subscript, is a value enclosed in parentheses immediately following the name of the array. In CC-40 BASIC an array begins with element 0.

The DIM Statement

To have the CC-40 reserve space for an array, specify the array in a DIM statement such as 130 DIM C(5) which reserves six locations, C(0) through C(5) for the array C. The CC-40 chooses a memory location, names it C, and reserves enough space for array C. You can use an array without including it in a DIM statement if you do not require more than 11 elements.

A One-Dimensional Array

Suppose the array C has had the following values assigned to the elements.

ARRAY C:	C(0)	C(1)	C(2)	C(3)	C(4)	C(5)
Value:	0	10	25	30	45	90

Then C(2) refers to the third element in array C, which has a value of 25 in this example. If M = 4, then C(M) refers to the fifth element of array C, which has a value of 45.

An example of a string array is shown below.

ARRAY NM\$:	NM\$(0)	NM\$(1)	NM\$(2)	NM\$(3)	NM\$(4)
Value:	Bob	Tom	Bud	Nancy	John

CHAPTER IV BASIC PROGRAMMING

A Two-Dimensional Array

You can extend an array to include information from a table which has rows and columns. A two-dimensional array has two subscripts that refer to the element's row and column. Suppose the array B is specified in the statement DIM B(2,2). Then the CC-40 reserves 9 locations for the 9 elements in array B. Suppose these locations have the values as shown below.

ARRAY B		
B(0,0) 15	B(0,1) 18	B(0,2) 21
B(1,0) 24	B(1,1) 27	B(1,2) 30
B(2,0) 33	B(2,1) 36	B(2,2) 39

Then you refer to the element of array B that has a value of 30 as B(1,2). Refer to the element of array B that has a value of 18 as B(0,1). You can refer to any element of array B as B(R,C), where R is equal to the row and C is equal to the column of the element.

In CC-40 BASIC you can have an array with up to 3 dimensions.

Using Arrays

Enter the following program in the CC-40. (Remember to type NEW and press [ENTER] before you start.)

```
100 DIM A(5),B(5)
```

Line 100 reserves six locations for array A and six for array B.

```
110 A(1)=2:A(2)=4:A(3)=6:A(4)=8:A(5)=10 !A(0) is not used
```

Line 110 has several assignment statements, assigning values to A(1) through A(5).

```
120 B(1),B(2),B(3)=2:B(4)=3E-5:B(5)=10 !B(0) is not used
```

Line 120 has several statements to store values in array B.

```
130 FOR C=1 TO 5
```

Line 130 sets up a loop which is executed 5 times.

```
140 PRINT "A*B = ";A(C)*B(C):PAUSE 2.5
```

Line 140 prints the string constant A*B= in the display, followed by the product of the array elements that the counter C refers to. The PAUSE statement holds the answer in the display for 2.5 seconds so you can see the answer.

CHAPTER IV BASIC PROGRAMMING

```
150 PRINT "A/B = ";A(C)/B(C):PAUSE 2.5
```

Line 150 prints the string constant A/B= followed by the quotient of the elements of the arrays and then pauses for you to read the answer.

```
160 NEXT C
```

Line 160 is the last line of the loop.

To list the program, type **LIST** and then press **[ENTER]** to see each succeeding line. To execute the program, enter **[RUN]**. The CC-40 displays the following products and quotients: 4 and 1; 8 and 2; 12 and 3; .00024 and 266666.6667; and 100 and 1.

READ and DATA Statements

When you have many values to assign to variables, you can easily assign them using **READ** and **DATA** statements. Each time a **READ** statement is executed, it reads data from a **DATA** statement. The values are written in a **DATA** statement(s) which may appear anywhere in your program.

A **READ** statement can read data into any number of variables. The data in the **DATA** statement is read from left to right. If necessary, a **READ** statement reads from more than one **DATA** statement. More than one **READ** statement can assign the values in a **DATA** statement; each **READ** statement assigns the first unread data value.

The previous example could use the **READ** and **DATA** statements instead of arrays and multiple assignment statements, as shown below.

```
100 DATA 2,2,4,2,6,2,8,3E-5,10,10
```

Line 100 lists the first value of A followed by the first value of B and then repeats values of A and B for all five pairs of numbers.

```
110 FOR C=1 TO 5
```

Line 110 sets up a loop to be executed five times.

```
120 READ A,B
```

Line 120 stores the first value from the **DATA** statement in A and the next value in B. Each time line 120 is executed, the next pair of values in the **DATA** statement is stored in A and B.

CHAPTER IV BASIC PROGRAMMING

```
130 PRINT "A*B = ";A*B:PAUSE 2.5
```

Line 130 prints A*B= followed by the product of A and B and then pauses 2.5 seconds for you to see the answer.

```
140 PRINT "A/B = ";A/B:PAUSE 2.5
```

Line 140 prints A/B= followed by the quotient of A and B and pauses 2.5 seconds.

```
150 NEXT C
```

Line 150 ends the loop.

To run this program, enter **[RUN]** and you will get the same answers as in the previous example.

READ and DATA with the RESTORE Statement

You can also cause a **READ** statement to assign values from the first **DATA** statement again or from other **DATA** statements by using the **RESTORE** statement.

The following program reads the first five pairs of values from the first **DATA** statement and reads the next two pairs from the second **DATA** statement. After the products and quotients have been displayed, the **RESTORE** statement in line 160 causes the next **READ** statement executed to start assigning values from the first value in the first **DATA** statement.

All the values in the two **DATA** statements are assigned and the sums printed. The **RESTORE** statement in line 210 causes the next **READ** statement executed to start assigning values from the first value in the **DATA** statement in line 105.

```
100 DATA 2,2,4,2,6,2,8,3E-5,10,10
```

```
105 DATA 30,40,5,20
```

```
110 FOR C=1 TO 7
```

```
120 READ A,B
```

```
130 PRINT "A*B = ";A*B:PAUSE 2.5
```

```
140 PRINT "A/B = ";A/B:PAUSE 2.5
```

```
150 NEXT C
```

```
160 RESTORE
```

```
170 FOR C=1 TO 7
```

```
180 READ A,B
```

```
190 PRINT A+B:PAUSE 2.5
```

```
200 NEXT C
```

```
210 RESTORE 105
```

```
220 READ A,B,C,D
```

```
230 PRINT A;B;C;D:PAUSE 2.5
```

CHAPTER IV BASIC PROGRAMMING

When you run this program, the following answers are displayed.

```
4
1
8
2
12
3
.00024
266666.6667
100
1
1200
.75
100
.25
4
6
8
8.00003
20
70
25
30 40 5 20
```

Expressions

In BASIC, calculations are performed by writing them as expressions. Expressions are constructed from constants, variables, and functions. There are four types of expressions: numeric, string, relational, and logical.

Numeric Expressions

A numeric expression is a series of one or more constants, variables, and/or functions connected by any of five arithmetic operators: +, -, *, /, ^ . An operator must appear between each pair of numeric constants, variables, and/or functions. When a numeric expression is evaluated, the result is always a number. CC-40 BASIC uses standard algebraic hierarchy in

CHAPTER IV BASIC PROGRAMMING

evaluating numeric expressions in the order given below. Refer to chapter 2 for a discussion of this order.

1. Calculations within parentheses are evaluated first.
2. Exponentiation is performed next.
3. Negation is performed next.
4. Multiplication/division from left to right is performed.
5. Addition/subtraction from left to right is performed last.

String Expressions

String expressions are constructed from string variables, string constants, and function references using the operation for concatenation (&) which combines or links strings. If the string length exceeds 255 characters, characters on the right are lost and the warning String truncation is displayed. The following is an example of string concatenation.

```
AB$="THIS IS AN EXAMPLE "
BB$="OF STRING CONCATENATION"
STRING$=AB$ & BB$
PRINT STRING$
```

Enter the above four lines in the CC-40 and it displays
THIS IS AN EXAMPLE OF STRING CONCATENATION.

Relational Expressions

Relational expressions are constructed from variables, constants, and functions to compare two values, using the following six relational operators.

- = (equal to)
- <> (not equal to)
- < (less than)
- <= (less than or equal to)
- > (greater than)
- >= (greater than or equal to)

The result of the comparison is either true or false. A relational expression has a value of -1 if it is true. A relational expression has a value of 0 if it is false.

Relational expressions are most often used in the IF THEN ELSE statement (described later in this chapter), but may be used anywhere that a numeric expression is allowed. The values compared must be either both numeric or both string.

CHAPTER IV BASIC PROGRAMMING

Numeric Comparisons

Relational expressions are evaluated from left to right after all arithmetic operations within the expression are completed. The following examples illustrate the use of relational expressions to compare numeric values.

150 IF X<Y THEN 200

If X is less than Y, control transfers to statement 200. If X is greater than or equal to Y, control continues with the statement after line 150.

200 A=2<5

210 PRINT A:PAUSE

Sets A equal to -1 since it is true that 2 is less than 5 and prints the value of A.

100 PRINT 2>5:PAUSE

Prints 0 since it is false that 2 is greater than 5.

String Comparisons

Comparisons of string values are performed by taking one character at a time from each string and comparing their ASCII codes. Leading and trailing blanks are significant. (See appendix D for a complete list of ASCII codes.) If the ASCII codes differ, the string with the lower code is less than the string with the higher code. If all the ASCII codes are the same, the strings are equal. For strings of unequal length, the comparison is performed for as many characters as there are in the shorter string. If all the ASCII codes are the same, the longer string is considered greater. The null string ("") is less than every other string.

100 PRINT "THIS" = "THAT":PAUSE

Prints 0 since it is not true that "THIS" is equal to "THAT".

110 PRINT "ABC"<"ABCD":PAUSE

Prints -1 since it is true that "ABC" is less than "ABCD".

Logical Expressions

The logical operators (AND, OR, NOT, and XOR) are generally used with relational expressions. The logical operators can also be used to manipulate data on a bit basis. Refer to appendix H for a description of using the logical operators in this way.

CHAPTER IV BASIC PROGRAMMING

The order of precedence for logical operators, from highest to lowest, is NOT, XOR, AND, and OR. The following examples illustrate the use of logical operators with relational expressions to form logical expressions. These logical expressions have a value of either true or false.

A logical expression with AND is true if the conditions on both its left and right sides are true.

100 IF 3<4 AND 5<6 THEN L=7

Sets L equal to 7 since 3 is less than 4 and 5 is less than 6.

110 IF 3<4 AND 5>6 THEN L=7

Does not set L equal to 7 because 3 is less than 4, but 5 is not greater than 6.

A logical expression with OR is true if either the condition on its left side is true, the condition on its right side is true, or both the conditions are true.

120 IF 3<4 OR 5>6 THEN L=7

Sets L equal to 7 because 3 is less than 4.

A logical expression with XOR (exclusive or) is true if either the condition on its left side is true, the condition on its right side is true, but not if both the conditions are true.

130 IF 3<4 XOR 5>6 THEN L=7

Sets L equal to 7 because 3 is less than 4 and 5 is not greater than 6.

140 IF 3<4 XOR 5<6 THEN L=7

Does not set L equal to 7 because 3 is less than 4 and 5 is less than 6.

A logical expression with NOT is true if the condition following it is not true.

150 IF NOT 3=4 THEN L=7

Sets L equal to 7 because 3 is not equal to 4.

Note: NOT 3=4 is equivalent to 3<>4.

160 IF NOT 3=4 AND (NOT 6=5 XOR 2=2) THEN 200

Does not pass control to line 200 because while it is true that 3 is not equal to 4, it is true that both 6 is not equal to 5 and 2 is equal to 2, so the condition in parentheses is not true.

Order of Execution of Expressions

The order of operations within arithmetic, relational, and logical expressions was given in the discussion for each type of expression. The order of precedence for evaluating expressions is given below.

- Functions are evaluated first.
- Arithmetic operations are performed next.
- String operations are performed next.
- Relational operations are performed next.
- Logical operations are performed last.

Input/Output Statements

Before data can be processed, it must be transferred into the computer. Data can be transferred into the computer by using assignment statements, READ and DATA statements, the KEY\$ function, or some form of input statement. In CC-40 BASIC the input statements are INPUT, LINPUT, and ACCEPT.

After the data has been processed, you either want to view it or store it for future use. To display or store the processed data, use some form of output statement. The output statements are PRINT and DISPLAY.

Using input and output statements with the display is described below. Using input and output statements with external devices such as printers and tape drives is described in this chapter under "Using External Devices".



The PAUSE Statement

The CC-40 displays printed items so quickly that you can not see them. There are three ways to have items remain in the display long enough so that you can read them.

First, you may have the computer pause after each statement that displays items by putting the statement PAUSE ALL in the program before any output statement. During a pause, the underline cursor is displayed in column 1 waiting for you to acknowledge the pause by pressing [ENTER] or [CLR]. After either of the keys is pressed, the computer resumes execution of the program with the next statement.

Second, you may have the computer pause after a specific statement by following that statement with PAUSE and the number of seconds that the pause is to last. In this case the cursor is not displayed. After the number of seconds specified has passed, the program resumes execution. If you use PAUSE with no time parameter, the cursor is displayed in column 1 and the program resumes execution after [ENTER] or [CLR] is pressed.

Finally, to keep a prompt for an ACCEPT statement in the display, you can follow the PRINT or DISPLAY statement with a comma or a semicolon to create a pending print (described later under "Pending PRINT and DISPLAY Statements").



Input Statements

The input statements allow a program to get data from the keyboard. The INPUT, LINPUT, and ACCEPT statements store the value(s) entered from the keyboard into the variable(s) listed in the statement. Only one value can be entered at a time. The KEY\$ function is used to halt program execution until a key is pressed.

Each statement is discussed briefly in the section below. Refer to chapter 5 for a detailed explanation of each input statement.

The INPUT Statement

You can put values typed on the keyboard into variables by using the INPUT statement. For example, enter the following program in the CC-40.

```
100 INPUT K
110 INPUT "ENTER DEGREES: ";D
120 INPUT A, B$
130 PRINT K;D;A;B$;PAUSE
```

When the program is run, the INPUT statement in line 100 halts program execution, displays a ? in column 1, and waits for a value to be entered from the keyboard. When a value is entered, it is stored in variable K.

Line 110 displays ENTER DEGREES: and waits for you to enter a value to be stored in D.

Line 120 displays a ? in column 1 and waits for you to enter a value for A. When a value is entered, it is stored in A. A question mark is displayed again to prompt you for a value for B\$.

The LINPUT Statement

The LINPUT statement assigns any series of characters entered from the keyboard to a string variable. Therefore, you can enter commas and leading and trailing spaces which are not allowed in the INPUT statement unless they are enclosed in quotes.

```
120 LINPUT "NAME: ";NEM$
```

displays NAME: and waits for a value to be entered that will be stored in NEM\$.

The ACCEPT Statement

The ACCEPT statement gives more control over data that is input from the keyboard. The options available in ACCEPT allow you to sound a tone, erase all or part of the display, limit the number and type of characters, and specify the column where they can be entered. For example, when the following line is executed, the computer beeps, erases the display, positions the cursor at column 10, and waits for you to enter a value with up to 4 characters for DEG. As you type the value, each character is tested to see if it is numeric (0-9, +, -, ., E) before it can be entered from the keyboard.

```
100 ACCEPT AT(10) VALIDATE(NUMERIC) BEEP ERASE ALL SIZE(4),
    DEG
```

The KEY\$ Function

The KEY\$ function allows you to halt program execution until a key is pressed. KEY\$ returns a one character string that corresponds to the key that was pressed. For example, when the following statement is executed, program execution halts until a key is pressed. The character corresponding to the key that was pressed is then stored in K\$. Refer to appendix D for a list of keycodes.

```
150 K$ = KEY$
```



Output Statements

The output statements allow you to write data in many different ways and on different media. Each statement is discussed briefly in the section below. Refer to chapter 5 for a detailed explanation of each output statement.

The PRINT and DISPLAY Statements

The PRINT statement allows you to print numbers and strings in the display. Negative values are preceded by a minus sign and nonnegative numbers are preceded by a space (instead of a plus sign). Numeric values are followed by a space.

When several values are to be displayed on a line, they are separated in the output statement with a semicolon or a comma. The semicolon causes the next value to be printed immediately after the preceding one. The comma causes the next value to be printed in the next field. The display is divided into fields 15 characters long. The fields start at columns 1, 16, 31, 46, 61, and 76.

The following statements print the output shown if X equals -7 and Y equals 13.

Statement	Output
100 PRINT X; Y:PAUSE	-7 13
110 PRINT X, Y:PAUSE	-7 13

You can also display string constants in an output statement. Unlike numeric values, string values have no leading signs and no trailing spaces. For example, the following statements print the output shown.

Statement	Output
180 X\$="X IS ";X=10:Y\$=" Y IS "; Y=20	
190 PRINT X\$,X;Y\$,Y:PAUSE	X IS 10 Y IS 20

The DISPLAY statement gives you more control than the PRINT statement over data that is displayed. The options available in DISPLAY allow you to sound a tone, erase the display, specify the size of items displayed, and specify the columns where

values are displayed. For example, when line 120 is executed, the computer beeps, erases the display, and displays the value of A\$ at column 3 followed by the value of B as shown below.

Statements	Output
110 A\$="The answer is ";B=15.55	
120 DISPLAY AT(3) BEEP ERASE ALL,A\$,B	The answer is 15.55
130 PAUSE	

USING with the PRINT and DISPLAY Statements

The PRINT and DISPLAY statements may optionally include a USING clause that allows you to display the numbers with a specific format. Pound signs (#) are used to show how many digits to use in printing the number. The format can be specified in the PRINT or DISPLAY statement itself or can be written in an IMAGE statement. For example, the following program uses the USING option in a PRINT statement.

```
100 INPUT "Enter Starting Mileage: ";SMILE
110 INPUT "Enter Ending Mileage: ";EMILE
120 INPUT "Enter Gallons Used: ";GALL
130 MPG=(EMILE-SMILE)/GALL
140 PRINT USING "Miles per gallon = ###.##";MPG
150 PAUSE
```

If you run this program and enter values of 5405.7, 5807.9, and 18.3, then Miles per gallon = 21.98 is displayed. Without the USING clause, the MPG would have been displayed as 21.97814208.

You could use the IMAGE statement by adding line 132 and changing line 140 as shown below.

```
132 IMAGE Miles per gallon = ###.##
140 PRINT USING 132;MPG
```

TAB with the PRINT and DISPLAY Statements

The TAB function is used with the PRINT and DISPLAY statements to format data, much as the TAB key on a typewriter does. TAB displays enough spaces to make the next value printed appear in a specific column.

The last example can be changed to use the TAB function to display values starting in a specific column. In the following

program, lines 134 and 136 have been added to use the TAB function.

```
100 INPUT "Enter Starting Mileage: ";SMILE
110 INPUT "Enter Ending Mileage: ";EMILE
120 INPUT "Enter Gallons Used: ";GALL
130 MPG=(EMILE - SMILE)/GALL
132 IMAGE Miles per gallon = ###.##
134 PRINT "Miles traveled: ";TAB(20);EMILE - SMILE:PAUSE 2.5
136 PRINT "Gallons used: ";TAB(20);GALL:PAUSE 2.5
140 PRINT USING 132;MPG
150 PAUSE
```

If you enter the same values as before, 5405.7, 5807.9, and 18.3, the display shows

```
Miles traveled:      402.2 (for 2.5 seconds)
Gallons used:        18.3 (for 2.5 seconds)
Miles per gallon =  21.98
```

Pending PRINT and DISPLAY Statements

The PRINT statements used in the examples have displayed exactly one line. Sometimes you may want to have several PRINT or DISPLAY statements display information on the same line. A pending print is created when a PRINT or a DISPLAY statement ends with a comma or semicolon. If a comma ends the statement, the computer spaces over to the next field; if a semicolon ends the statement, the computer does not space over. Then the next PRINT or DISPLAY statement prints on the same line at the current column position.

The program above can be changed to print the mileage and the gallons on the same line by changing lines 134 and 136 as shown below.

```
100 INPUT "Enter Starting Mileage: ";SMILE
110 INPUT "Enter Ending Mileage: ";EMILE
120 INPUT "Enter Gallons Used: ";GALL
130 MPG=(EMILE - SMILE)/GALL
132 IMAGE Miles per gallon = ###.##
134 PRINT "Miles = ";EMILE - SMILE;
136 PRINT "Gallons = ";GALL:PAUSE 2.5
140 PRINT USING 132;MPG
150 PAUSE
```

Control Statements

Most of the programs you have run on the CC-40 started executing the first statement and continued executing each sequential line to the last. The flow of the program or flow of control has gone from the first statement to the last.

Control statements are used to direct the flow of the program. Some statements form a loop and cause some lines to be repeated a specified number of times. You have already used two of these statements, the FOR TO STEP and NEXT statements. Some statements compare data and cause program execution to jump or branch to another line rather than go to the next program statement. This section describes the various control statements available in CC-40 BASIC.

The FOR TO STEP Statement

You have already used the FOR TO and NEXT statements in a program to create a loop. The FOR TO statement has another option that allows you to increment the counter other than by 1. For example, entering

```
100 FOR COUNT = 2 TO 100 STEP 2
```

starts a loop where the counter begins at 2 and is incremented by 2 each time. The loop is repeated until the counter is greater than 100.

If the starting value of the counter is greater than the limit value, the loop is not executed. If the starting value and the limit value of the counter are the same, the loop is executed one time.

You can also use a negative value for STEP. The counter of the FOR statement is decreased each time the loop is executed. If the starting value of the counter is less than the limit value, the loop is not executed. If the starting value and the limit value of the counter are the same, the loop is executed one time.

Enter the following program. The CC-40 displays the steps it calculates in the loop.

```
100 FOR A = 6 TO 4 STEP -.25
110 DISPLAY AT(10) BEEP,"A = ";A:PAUSE 2.1
120 NEXT A
```


You should not transfer control into the middle of a loop from the outside. The counter or control variable is set up only when the FOR TO statement is executed. You may transfer control out of a loop with a GOTO, GOSUB, ON GOTO, or ON GOSUB statement and then transfer back in.

Nested Loops

A FOR TO NEXT loop can be contained within another loop. The loop that is inside is called a nested loop. A nested loop must always be entirely inside the outer loop. For example, in the program above, the value of the counter can be displayed in successive columns by adding statements 105 and 130 and by changing statements 110 and 120 as shown below.

```
100 FOR A=6 TO 4 STEP -.25
105 FOR B=1 TO 7 STEP 3
110 DISPLAY AT(B) BEEP,"A= ";A:PAUSE 2.1
120 NEXT B
130 NEXT A
```

The GOTO Statement

The GOTO statement tells the computer what line in a program to execute next. The following program uses a GOTO statement to read all the data in the DATA statement.

```
100 DATA 5,10,3.5,420,55.25
110 READ R
120 PRINT R, 2*PI*R:PAUSE 2
130 GOTO 110
```

Each time line 130 is executed, control is transferred back to line 110 which is executed again. When line 110 tries to read past the data in the DATA statement, an error occurs and the message DATA error is displayed. To determine when to stop reading data, a dummy value (a value you know marks the end of the data) can be inserted in the DATA statement and the IF THEN ELSE statement used to test it.

The IF THEN ELSE Statement

The IF THEN ELSE statement allows you to compare data in a program. The data compared can be constants, variables, and/or expressions. If the comparison or condition being tested is true, the statement(s) following the word THEN are executed. If the comparison is false, the statement(s) following the word ELSE are executed. If the comparison is false and there is no ELSE, the line following the IF statement is executed.

In the following program a check is made on the data read. If the dummy value (a value less than zero) has been read, an end of data message is printed. If the dummy value has not been read, the result of the calculation is printed. After [ENTER] is pressed, the next value in the DATA statement is read.

```
100 PAUSE ALL
110 DATA 5,10,3.5,420,55.25, -5
120 READ R
130 IF R<0 THEN 160
140 PRINT R, 2*PI*R
150 GOTO 120
160 PRINT "END OF DATA"
```

The following are examples of IF THEN ELSE statements.

```
400 IF D=999 THEN DISPLAY "ARE YOU FINISHED?" ELSE 150
```

The computer checks the value in location D to determine if it is 999. If D is 999, the computer displays ARE YOU FINISHED? and executes the next line. If D is not 999, the computer executes line 150.

```
510 IF L(C) <> 12 THEN C=S+1 ELSE COUNT=COUNT+1:GOTO 140
```

The computer checks the value in L(C) and if it is not equal to 12, then C is set equal to S+1 and the next line is executed. If L(C) is equal to 12, then COUNT is set equal to COUNT plus 1 and line 140 is then executed.

The ON GOTO Statement

Another control statement is ON GOTO. The ON GOTO statement is used to transfer control to a program line based on whether the value of the variable following the word ON is 1, 2, 3, etc.

CHAPTER IV BASIC PROGRAMMING

```
100 REM THIS PROGRAM IS A DEMONSTRATION
110 I OF THE ON GOTO STATEMENT
120 PRINT "1 for LOG, 2 for LN, 3 for EXP";
130 ACCEPT AT(31) SIZE(1) BEEP VALIDATE("123"),CODE
140 DISPLAY ERASE ALL, "ENTER ARGUMENT:";
150 ACCEPT BEEP, ARG
160 IF ARG < 0 THEN 140
170 ON CODE GOTO 180, 200, 220
180 PRINT "LOG of ";ARG;"is ";LOG(ARG):PAUSE
190 GOTO 120
200 PRINT "LN of ";ARG;"is ";LN(ARG):PAUSE
210 GOTO 120
220 PRINT "EXP of ";ARG;"is ";EXP(ARG):PAUSE
230 GOTO 120
```

The program above accepts a 1, 2, or 3 for the variable CODE. The PRINT statement displays a prompt and the ACCEPT statement halts program execution until a value is entered for ARG. If the value of ARG is negative, the prompt is again displayed and the ACCEPT statement waits for another value for ARG. When a nonnegative value is entered for ARG, the program calculates the LOG, LN, or EXP of ARG depending upon the value entered for CODE.

Strings and String Manipulation

String constants and string variables have already been defined in this chapter. However, you may find that you need to be able to manipulate a string. This section describes strings and the functions you can use on the CC-40 to manipulate them.

Each character is stored in the CC-40 as a number from 0 through 255. The number is called the ASCII character code. For example, the string values "BASIC" and "Basic" are represented as shown below. The string BASIC is less than the string Basic because the ASCII code for A is less than the ASCII code for a.

B	A	S	I	C	B	a	s	i	c
66	65	83	73	67	66	97	115	105	99

CHAPTER IV BASIC PROGRAMMING

Converting a Character to ASCII Code—ASC

You can convert the first character in a string to its ASCII character code by using ASC.

```
100 NUMB1 = ASC("H")
110 NUMB2 = ASC("hello")
120 NUMB3 = ASC("%")
```

Assign NUMB1 the value 72, NUMB2 the value 104, and NUMB3 the value 37.

Converting a Number to its Corresponding Character—CHR\$

You can convert a number (from 0 through 255) to the character that is designated by the number according to ASCII conventions.

```
100 A$ = CHR$(42)
```

Assigns the character * to A\$.

The following program accepts a value from the keyboard. If the value is a lower-case character, it is changed to upper-case. The value is then printed and control returns to statement 100. To terminate the program, press [BREAK].

```
100 PRINT "Press a key: "
110 A$ = KEY$
120 IF ASC(A$) < 97 OR ASC(A$) > 122 THEN 140
130 A$ = CHR$(ASC(A$) - 32)
140 PRINT "The character is now ";A$:PAUSE 2:GOTO 100
```

Finding the Length of a String—LEN

You can determine the length of a string by using the LEN function.

```
100 ALB$ = "LIST 10"
110 N = LEN(ALB$)
```

Define the string ALB\$ and assign the number of characters in ALB\$ to the variable N. In this case N has a value of 7.

Repeating a String—RPT\$

You can repeat a string by using the RPT\$ function.

```
130 PRINT RPT$("BASIC ",5):PAUSE
```

Displays the string BASIC BASIC BASIC BASIC BASIC .

Finding a String within a String—POS

You can determine the position of one string within another string by using the POS function.

```
140 BB = POS(ALB$,"ST",1)
```

Assigns the variable BB the position in string ALB\$ where the string "ST" first occurred. From the example above, ALB\$ is equal to "LIST 10" and the position where the string "ST" first occurs in ALB\$ is 3.

Getting a Substring of a String—SEG\$

You can get a substring of a string by using the SEG\$ function.

```
190 CC$ = SEG$(ALB$,4,3)
```

Assigns the variable CC\$ three characters from the string ALB\$ starting at the fourth character. If ALB\$ is equal to "LIST 10", CC\$ is set equal to T 1.

Converting a Number to a String—STR\$

You can convert a number to a string by using the STR\$ function.

```
150 BB = 17
```

```
160 VALUE$ = STR$(BB)
```

Converts the number in BB to a string that represents that number and assigns it to VALUE\$. In this case VALUE\$ contains the string "17".

Converting a String to a Number—VAL

You can convert a string to a number by using the VAL function.

```
170 NOMBRE = VAL(VALUE$)
```

Converts the string representing a number in VALUE\$ (which is "17") to the number and assigns it to NOMBRE. In this case, NOMBRE has a value of 17.



Testing a String for a Numeric Constant—NUMERIC

You can test a string to determine if it is a valid representation of a numeric constant by using the NUMERIC function.

NUMERIC returns a value of -1 (true) if the string is a valid representation of a numeric constant and a value of 0 (false) if it is not. NUMERIC can be used on a string to see if VAL will convert it to a numeric value.

The statements below are used to test if A\$ is a valid representation of a numeric string before the VAL function is used to change the string to a number.

```
160 IF NUMERIC(A$) THEN A = VAL(A$) ELSE PRINT "NOT A  
NUMBER":PAUSE
```

Built-in BASIC Functions

All of the CC-40 BASIC functions may be used in a program line. The trigonometric and logarithmic/exponential functions have been discussed in chapter 2. Some CC-40 functions are described in this chapter in sections that deal with similar instructions. The functions available on the CC-40 to manipulate numbers and generate random numbers are described below.

Manipulating Numbers

The absolute value of an expression can be obtained by using ABS. In the example below, K is set equal to 20. ABS always returns a positive value or zero.

```
100 K = ABS(-4*5)
```

The sign of a number can be determined by using SGN. In the example below, K is set equal to 1 if C is positive, 0 if C is zero, and -1 if C is negative.

```
110 K = SGN(C)
```

The INT function is used to find the largest integer that is less than or equal to a number. In the example below, K is set equal to 23 and L is set equal to -5.

```
120 K = INT(23.99999)
```

```
130 L = INT(-4.1)
```


Generating Random Numbers

You can have the CC-40 generate random numbers for programs involving statistical analysis, games, and simulations. The CC-40 produces random numbers from 0 to 1 when RND is used. For example, the program below generates 10 random numbers.

```
100 FOR J=1 TO 10
110 PRINT RND:PAUSE 1.5
120 NEXT J
```

The same series of random numbers is generated each time you run a program unless a RANDOMIZE statement is executed before generating the random numbers.

The program below prompts for the number of random numbers to be generated. The random numbers are printed one at a time followed by their average. Press [BREAK] to stop the program.

Note: The average of many random numbers is approximately .5.

```
100 INPUT "ENTER NUMBER OF VALUES: ";QUAN
110 AVER=0
120 FOR A=1 TO QUAN
130 D=RND:PRINT D:PAUSE 2
140 AVER=AVER+D
150 NEXT A
160 PRINT "Average is ";AVER/QUAN:PAUSE
170 GOTO 100
```

To generate a sequence of integer random numbers, you can use INTRND. INTRND generates a random number between 1 and the number that you give it. For example, the program below generates ten random numbers between 1 and 100.

```
100 FOR J=1 TO 10
110 PRINT INTRND(100):PAUSE 1.5
120 NEXT J
```

Subroutines

Many times in a program you may find that you need to use the same group of lines in several places. By writing these lines as a subroutine, you can eliminate the need to duplicate them. Then when you need to execute these lines, you transfer control to the subroutine. When the subroutine has finished executing, control is transferred back.

The GOSUB Statement

To transfer control to a subroutine, you can use the GOSUB statement followed by a line number. The line number is the first program line in the subroutine. The last line of a subroutine must be a RETURN statement that transfers control back to the statement after the GOSUB statement. A subroutine may also transfer control to another subroutine, allowing nesting of subroutines. Refer to chapter 5 for a description of GOSUB.

A subroutine can use and change the values of any variables in the main program which includes it.

When the GOSUB statement is executed, the following process takes place.

1. A pointer to the statement after the GOSUB statement is stored by the computer.
2. Program control transfers to the line specified by the GOSUB statement.
3. The statements of the subroutine are executed.
4. Program control transfers to the address contained in the pointer when the RETURN statement is encountered.
5. Execution resumes with the statement following the GOSUB statement.

The ON GOSUB Statement

The ON GOSUB statement is another way to call a subroutine. ON GOSUB determines which subroutine to call according to the value of the variable following the word ON.

CHAPTER IV BASIC PROGRAMMING

100 ON XVAL GOSUB 200, 400, 600

Transfers control to the subroutine at line 200 if XVAL is 1, to line 400 if XVAL is 2, and to line 600 if XVAL is 3.

XVAL is rounded to the nearest integer.

Subprograms

Like subroutines, subprograms eliminate the need to write duplicate program lines. However, subprograms operate very differently from subroutines. Subprograms are executed by using the CALL statement followed by the subprogram's name and, optionally, a list of arguments enclosed in parentheses. When a program includes subprograms, they must follow the main program.

CC-40 BASIC Subprograms

The first statement in a subprogram must be the SUB statement followed by an optional list of parameters. If a subprogram needs data from the main program, the data must be passed through the parameters. The variables in a main program are restricted for use by the main program and any subroutines in the main program. The variables in a subprogram are restricted for use in the subprogram and any subroutines within the subprogram. Therefore, variable names may be duplicated in a main program and a subprogram. A subprogram may call other subprograms, but must not call itself, either directly or indirectly.

The last statement in a subprogram must be a SUBEND. When the SUBEND statement is executed, control returns to the statement following the CALL statement that called the subprogram. Control may also be returned by the SUBEXIT statement before the end of the subprogram.

Argument List

Data is passed to the subprogram through the argument list of the CALL statement. Each argument in the argument list has a corresponding variable in the parameter list of the subprogram. The arguments of the argument list can be constants, variables, arrays, or expressions. Arguments can be passed by reference or by value.

CHAPTER IV BASIC PROGRAMMING

Passing Arguments By Reference

Arguments that are passed by reference can be variables or arrays. Arrays are always passed by reference. When arguments are passed by reference, the subprogram uses those variables or arrays from the calling program. If a parameter in a subprogram is changed, its corresponding argument in the calling program is also changed.

In the program segment below, the subprogram uses the variables A, B, and L(3) and can change the stored values of these variables. When the SUBEND statement at line 990 is executed, program control returns to line 210 in the main program.

```
200 CALL MARINE(A,B,L(3))
210 ...
...
900 SUB MARINE(F,L,Z)
...
990 SUBEND
```

Passing Arguments By Value

Arguments that are passed by value can be variables, constants, or expressions. To pass a variable by value, the variable must be enclosed in a set of parentheses. Constants and expressions are always passed by value. When arguments are passed by value, the subprogram can use the values of the arguments, but the subprogram cannot alter the values of the arguments.

In the program segment below, the values of the variables A and L(3) and the expression L(3) + 1 are passed by value to the subprogram TRACTION, along with the constant 17. TRACTION cannot alter the values of the variables A and L(3) in the calling program or subprogram, but it can alter the value of the variable B. When the SUBEND statement at line 990 is executed, program control returns to line 210 in the calling program or subprogram.

```
200 CALL TRACTION((A),B,(L(3)),17,L(3) + 1)
210 ...
...
900 SUB TRACTION(N,L,Z,D,P)
...
990 SUBEND
```

The ATTACH and RELEASE Statements

You can reduce the execution time of a program that repeatedly calls a subprogram by using the ATTACH statement when you have sufficient free memory. When a subprogram is attached, the variables are initialized when the ATTACH is executed and not each time the subprogram is called. The values of the variables are maintained when the subprogram terminates.

To release the memory that is used when a subprogram is attached, use the RELEASE statement. The variables in the subprogram are then initialized each time the subprogram is called, and are not maintained when the subprogram terminates.

Built-in Subprograms

The CC-40 has many built-in subprograms that you can access. The following sections describe these subprograms.

Expanding Memory

You can add to the internal memory of the CC-40 by using CALL ADDMEM. CALL ADDMEM appends the Random Access Memory (RAM) in the *Memory Expansion* cartridge to resident memory. See chapter 3 and appendix J for a description of memory expansion.

Using Memory

The CC-40 provides the capability of using assembly language programs and subprograms. The function FRE can be used to determine the amount of memory available and the following BASIC subprograms can be used to access it: GETMEM, POKE, PEEK, EXEC, LOAD, IO, and RELMEM. Refer to chapter 5 for more information.

The FRE Function

The FRE function is used to determine how much memory is being used for the operating system and the program in memory and how much memory is available.

The GETMEM Subprogram

The GETMEM subprogram is used to reserve the memory that you have determined is available from the FRE function. You can store data and assembly language programs and subprograms there. The amount of memory reserved should be significantly less than the largest block available. Sufficient memory space must remain available for statements that require additional temporary memory. GETMEM requires four bytes of memory for its own operation.

The POKE Subprogram

The POKE subprogram is used to write data or an assembly language program or subprogram in reserved memory. If you use POKE indiscriminately, you may erase programs and/or files. You cannot do any physical harm to the CC-40 with POKE, but you may have to reset the system.

The PEEK Subprogram

The PEEK subprogram is used to read the data in memory locations.

The LOAD Subprogram

The LOAD subprogram is used to load into computer memory an assembly language subprogram from external storage. More than one subprogram may be loaded into memory and if space permits, they may reside in memory with a BASIC program.

The EXEC Subprogram

The EXEC subprogram is used to execute an assembly language program or subprogram.

The IO Subprogram

The IO subprogram is used to perform control operations on peripheral devices.

The RELMEM Subprogram

The RELMEM subprogram is used to release the memory you reserved with GETMEM.

CHAPTER IV BASIC PROGRAMMING

Language Prompting

You can use SETLANG to display system messages in either English or German. Many of the *Solid State Software*™ cartridges provide messages in languages in addition to English. By using GETLANG, you can determine which language is currently in use. The statement below sets the language code to German.

```
CALL SETLANG(1)
```

To reset the language code to English, enter CALL SETLANG(0).

Display Assignments

With CALL CHAR you can define your own displayable characters. With CALL INDIC you can turn the display indicators on and off.

The CHAR Subprogram

CALL CHAR can define up to 7 displayable characters at one time. In the example below, character codes 1 and 2 are defined to be up and down arrows by using CALL CHAR. For a description of CALL CHAR, refer to chapter 5.

```
100 CALL CHAR(1,"040404041F1F0E04")
110 CALL CHAR(2,"040E1F1F04040404")
```

The INDIC Subprogram

There are 17 indicators in the display that you can turn on and off. The six indicators at the bottom of the display are reserved for your use. If the other indicators are turned on and off, undesirable results may occur. For more information on CALL INDIC, refer to chapter 5.

To turn on indicator one, the following statement can be used.

```
130 CALL INDIC(1)
```

The program below uses both CALL CHAR and CALL INDIC. CALL CHAR is used to define up and down arrows for character codes 1 and 2. The user is allowed six chances to guess the number the computer has stored. Each time a wrong guess is made, one of the six indicators in the display is turned on by CALL INDIC. When the number is guessed or when the six chances have been used, a message is printed.

CHAPTER IV BASIC PROGRAMMING

```
100 CALL CHAR(1,"040404041F1F0E04")
110 CALL CHAR(2,"040E1F1F04040404")
120 DISPLAY BEEP,"GUESS A NUMBER BETWEEN 1 AND
    25":PAUSE 1.5
130 DISPLAY BEEP,"YOU HAVE 6 CHANCES":PAUSE 1
140 DISPLAY BEEP,"INDICATORS RECORD YOUR
    GUESSES":PAUSE 1
150 COUNT = 1
160 RANDOMIZE
170 SNUM = INTRND(25)
180 PRINT "ENTER YOUR GUESS ";
190 ACCEPT AT(28)VALIDATE(DIGIT),GUESS
200 IF GUESS = SNUM THEN 280
210 CALL INDIC(COUNT)
220 IF COUNT = 6 THEN 360
230 IF GUESS < SNUM THEN 260
240 PRINT CHR$(1);GUESS;"Try a smaller number";
250 COUNT = COUNT + 1:GOTO 190
260 PRINT CHR$(2);GUESS;"Try a larger number";
270 COUNT = COUNT + 1:GOTO 190
280 PRINT "WOW!!***";GUESS;"*** is correct":PAUSE 2
290 FOR A = 1 TO 0 STEP - 1:X = 1
300 FOR B = 1 TO 6
310 CALL INDIC(B,A):PAUSE .1
320 DISPLAY AT(X)BEEP,"YEA!":X = X + 5
330 NEXT B
340 NEXT A
350 GOTO 420
360 PRINT "6 Chances!";SNUM;"was the number":PAUSE 4
370 FOR A = 1 TO 30 STEP 10
380 DISPLAY AT(A) BEEP,"BOO!":PAUSE .2:NEXT A
390 FOR J = 1 TO COUNT
400 CALL INDIC(J,0)
410 NEXT J
420 PRINT "Press ENTER to play again":PAUSE
430 GOTO 150
```


The KEY Subprogram

The KEY subprogram is used to determine which key, if any, is pressed. For example, CALL KEY(K,S) assigns to K the ASCII code of the current key that is pressed. S is set equal to 1 if the key pressed is different from the one the last time the KEY subprogram was called. S is set equal to -1 if the same key is pressed that the last call to KEY returned, and to 0 if no key is pressed.

For example, the following section of a program waits for a key to be pressed and then checks if the key was Y or y.

```
:  
150 CALL KEY(K,S)  
160 IF S=0 THEN 150  
170 IF K=ASC("y") or K=ASC("Y") THEN 250  
:
```

The VERSION Subprogram

The VERSION subprogram is used to determine the version of BASIC that is being used. CALL VERSION(V) sets V equal to 10, the BASIC used on the CC-40.

The CLEANUP Subprogram

You can eliminate any variables that are not being used in the current program in memory by calling the subprogram CLEANUP. CLEANUP cannot be called from a program.

The DEBUG Subprogram

The DEBUG subprogram is used to access the debug monitor to allow you to read and change memory locations and run and debug your assembly language programs and subprograms. Refer to appendix I for a description of DEBUG.

Handling Errors in a BASIC Program

The CC-40 provides a means of processing errors in a BASIC program by using the ON ERROR statement and the ERR subprogram. When a program is executed, the error handler is automatically set to display a message and stop program execution when an error occurs. However, you can modify the error handler to cause it to execute a subroutine when an error occurs by using ON ERROR followed by a line number.

The line number must be the beginning of a subroutine. In the subroutine, you can call the ERR subprogram to obtain the error code of the error that occurred. You can then compare error codes in the subroutine and determine what caused the error. The subroutine must end with a RETURN statement. Refer to appendix K for a complete list of the error codes.

For example, in the following program, the ON ERROR statement causes any errors that occur to be handled by the subroutine starting at line 300. The program accepts the name of the next program to run. The computer searches for the program. If the program is not found, an error occurs. The subroutine prints a prompt for another program name to be entered. The program continues execution when the program to be executed is found. If the error occurred for any other reason, the error code and the line number are printed and the program stops.

```
190 ON ERROR 300  
200 INPUT "ENTER PROGRAM NAME ";PROG$  
210 RUN PROG$  
300 REM ERROR HANDLING SUBROUTINE  
310 CALL ERR(CODE,TYPE,FILE,LINE)  
320 IF LINE<>210 THEN RETURN 360  
330 IF CODE<>15 THEN RETURN 360  
340 PRINT "Prog. not found, press CLR":PAUSE  
350 RETURN 190  
360 REM PRINT ERROR SUBROUTINE  
370 PRINT "ERROR";CODE;" IN LINE";LINE:PAUSE
```

Handling Breaks in a BASIC Program

The CC-40 provides a means of processing breakpoints that occur in a BASIC program. When a program is executed, the computer automatically halts program execution and displays a message when a breakpoint occurs. However, by using ON BREAK, you can cause breakpoints to be ignored (including the [BREAK] key) or to be treated as errors. If breakpoints are treated as errors, the ON ERROR statement can process them as described above. See ON BREAK in chapter 5 for more information.

Handling Warnings in a BASIC Program

The CC-40 provides a way to handle warnings that occur in a BASIC program. When a warning occurs while a program is executing, the computer automatically displays a warning message and then continues program execution when [CLR] or [ENTER] is pressed. However, by using ON WARNING, you can cause a warning message not to be displayed or a warning to be treated as an error.

Debugging

You may find that a program does not work the way you intended. The errors that are in it are logical errors, called "bugs" in computer usage. Testing a program to find these bugs is called "debugging" a program.

Finding Bugs

Remember that a program is doing exactly what it was told to do. When the program is not working properly, think about what could be going wrong, then devise tests to perform within the program to aid you in finding the bugs.

Debugging Aids

When you are debugging a program, you can use the following aids to help track down the error.

The [BREAK] key stops program execution when the key is pressed. At this point you can clear the break message and print or change the values of variables.

The BREAK statement allows you to stop a program at specific lines to determine what is happening in the program. You can print or change the values of the variables.

130 BREAK

Stops the program when the BREAK statement is executed.

230 BREAK 240,250

Sets breakpoints immediately before lines 240 and 250.

BREAK 300,350,380

Can be entered for immediate execution either before you RUN a program or while you are in the middle of a break to set breakpoints before lines 300, 350 and 380.

The CONTINUE command causes the computer to continue program execution after a breakpoint. Press [CLR] and type CONTINUE (or CON) and press [ENTER].

The UNBREAK statement is used to remove the breakpoints you have set in a program. The only breakpoints that are removed are the ones that are set immediately before a line. In lines 130 and 230 in the previous example, UNBREAK can only remove the breakpoints set before lines 240 and 250. Line 130 always halts program execution when it is executed.

Using External Devices

Programs can be saved on external devices and then reloaded into memory and run. Data can be stored on external devices such as the TI *Wafertape*™ peripheral and programs created to update this data. External devices such as printers can be used to provide information in a form you can read. When the computer is transmitting data to or receiving data from an external device, the I/O display indicator is turned on. You cannot use the keyboard at this time (including the [OFF] key).

If a file is open when you press the [OFF] key, the file is automatically closed before the computer is turned off. (Generally the term file refers to data stored on a mass storage device. However, in CC-40 BASIC a file refers to any information sent to an external device, even a printer.)

You can save programs on an external device and later run these saved programs by using the SAVE and OLD commands and the RUN statement. SAVE writes the program in memory to an external device in the internal machine format. The OLD command is used to load a SAVED program into memory when you want to edit the program or VERIFY that it was loaded correctly. To execute the program, use the RUN statement.

Note: If you attempt to load (with OLD) or RUN a data file rather than a program file, you may have to press the reset key to reset the system.

You can list a program to an external device such as a printer by using the LIST command. LIST writes the program in memory to an external device in ASCII characters, the same form you see in the display.

CHAPTER IV BASIC PROGRAMMING

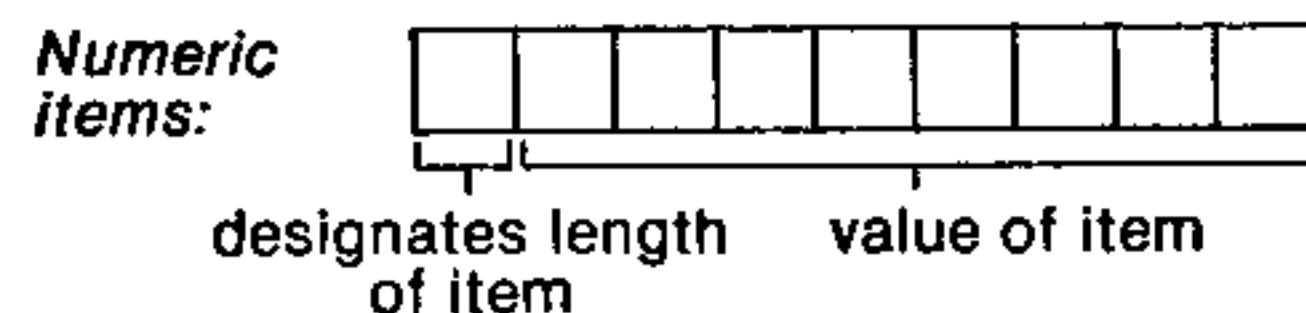
You can store, update, and print data to an external device by using the BASIC input/output statements. You must first open a file on an external device with the OPEN statement before you can use a BASIC input/output statement to access the file. The OPEN statement is used to inform the computer how the data on the file is stored and the number that you will use to access the file. You do not use the OPEN statement when you use the BASIC commands, SAVE, OLD, or LIST.

Data Format

If data is to be stored, updated, or printed, you must specify to the computer the data format or how the data is recorded. When data is printed for people to read, the data should be written in ASCII characters (like the characters you see in the display). This type of data format is called display. When display-type data is printed, the numeric and string items are written according to the specifications in PRINT and appear the same as if the items were displayed in the CC-40.

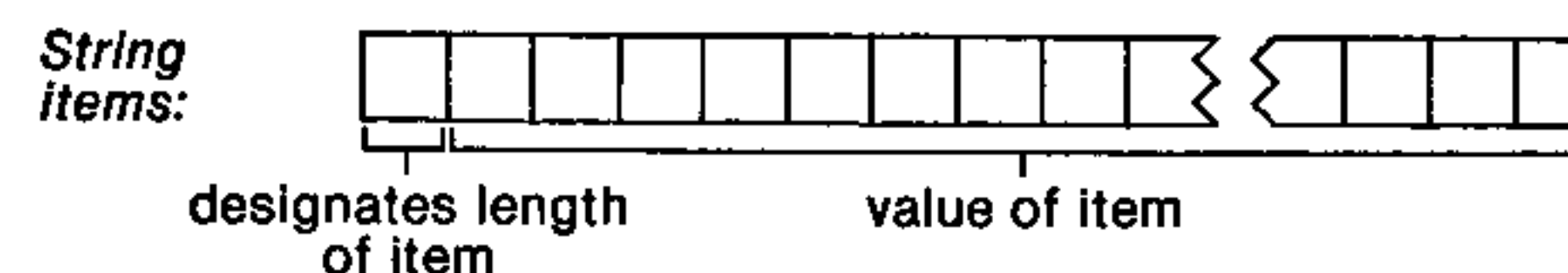
If the data is stored on a mass storage device, the data should be recorded in the internal machine-code format. Data written in this internal-type format is stored in binary code, the type the computer uses to process data. Storing data in this format expedites processing and reduces the storage space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

- Numeric items are stored in a form which occupies from 3 through 9 bytes of memory. The first byte is used to store the length of the numeric data item and the remaining 2 through 8 bytes are used to store the data value.



CHAPTER IV BASIC PROGRAMMING

- String data is stored in the same manner, except that the maximum length for a string item extends through 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.



Data Records

Data is 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. After a file is created, the CC-40 retrieves and updates data from the file in terms of records.

Record Length

When you write records to an external device, you can specify the *maximum* length for the records. If you do not specify a maximum record length, the computer assumes a default value according to the peripheral device you are using. 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.

The record length you specify determines how much space is reserved in the computer for storing a record of the file. If you attempt to write a record that is longer than the record length you specified, the computer breaks the record into smaller parts as described in PRINT (with files) in chapter 5. 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.

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 you read it in sequence from the beginning, the file should be organized sequentially. Data stored in a sequential file is read the same as you would read data 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 specify that a file is a relative one 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, LINPUT, PRINT, and RESTORE statements. Relative files can also be accessed sequentially. Only certain types of devices support relative files.

File Processing Keywords

CC-40 BASIC provides an extensive range of file-processing features including sequential and random file organization and processing, variable length records, display and internal data formats, and file accessibility. This section describes the CC-40 BASIC keywords which are provided to facilitate file processing—FORMAT, DELETE, OPEN, INPUT, LINPUT, PRINT, CLOSE, EOF, and RESTORE. Refer to chapter 5 for a complete description of these keywords.

The FORMAT Command

The FORMAT command initializes the medium on an external storage device. You must format a new medium (such as a tape) before writing on it. If you format a medium that has data already written on it, all the data is erased. For example, the command below formats the tape on device 1.

```
FORMAT 1
```

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 "2.MFILE"
```

Deletes the file named MFILE on device 2.

The OPEN Statement

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 (except how the file can be accessed).

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. Since 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.

CHAPTER IV BASIC PROGRAMMING

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.
<u>Open-Mode Attribute</u>	<u>File Accessibility</u>
INPUT	The computer can only <i>read</i> from the file.
OUTPUT	The computer can only <i>write</i> to the file.
UPDATE	The computer can both <i>read</i> from and <i>write</i> to the file.
APPEND	The computer can write data <i>only</i> at the end of the file. The records that already exist on the file cannot be accessed.
File Organization:	RELATIVE for random access file or omitted for sequential file.
File Types (Data Formats):	DISPLAY or INTERNAL. If file type is omitted, DISPLAY is assumed.
Record Length:	VARIABLE followed by a numeric expression for the record length. If this option is omitted, the maximum record length is established by the peripheral device.

CHAPTER IV BASIC PROGRAMMING

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 file-name AFILE. The attributes of the file are relative (random access) organization, internal data format, INPUT open-mode, and a maximum record length of 64 bytes.

```
100 OPEN #5,"100.AFILE", RELATIVE, INTERNAL, INPUT,  
      VARIABLE 64
```

The statement below opens a file named BFILE on device 1 as #7. The file can be both read from and written to (UPDATE open-mode), has sequential organization, and is recorded in display-type data.

```
150 OPEN #7,"1.BFILE"
```

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 is 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 opened file.

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 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.

```
100 OPEN #3,"1.MYFILE",INTERNAL,VARIABLE 64
110 INPUT #3,A$,J,K,L,B$,P,Q,R
```

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 fields left in the input buffer are retained. When the next INPUT statement is executed, the remaining fields are assigned to the variables.

```
100 OPEN #3,"1.MYFILE",INTERNAL,VARIABLE 64
110 INPUT #3,A$,J,K,L,B$,P,Q,R,
120 INPUT #3,C$,A,B,C
```

Input # and Data Formats

When the INPUT statement reads display-type data, the fields are separated by the commas that appear between the fields. Display-type records look like the data in a DATA statement. Numeric and string items must appear with their separators. Each field in a display-type record is checked to ensure that numeric values are placed in numeric variables.

In the example below, the first time the INPUT statement is executed, it assigns the fields in the first record to the variables. The second time the INPUT statement is executed, the fields in the second record are assigned to the variables. Note that there is no field in the buffer for the last variable, so the next record is read. When the INPUT statement attempts to assign a field to the last variable, an error occurs. The variable is a numeric variable but the field is not a numeric value.

```
(Record #0 on file #3) Jones, 95,98,65,32,78
(Record #1 on file #3) Smith, 67,87,66,90
(Record #2 on file #3) Lee,89,88,90,67,90
```

```
100 OPEN #3,"1.MYFILE",INTERNAL,VARIABLE 64
110 INPUT #3,NAMES,A,B,C,D,E
120 GOTO 110
```

When the INPUT statement reads internal-type data, the length byte stored with each data item is used to separate the fields. The only validation performed on internal-type data is to ensure that numeric data is from 2 through 8 characters long.

The LINPUT # Statement

Like the INPUT # statement, the LINPUT # statement is used to read records from a file. However, LINPUT places all commas, leading and trailing spaces, semicolons, and quotation marks into string variables. The INPUT statement places these symbols into variables only if they are enclosed in quotation marks.

The PRINT # Statement

The PRINT # statement writes data values to a file. You must use the same file-number in opening and writing to 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). For UPDATE mode you must first read to the end of a 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. If the PRINT statement ends with a comma or a semicolon, a pending print is created.

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 written 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,"1.PENDING",INTERNAL,OUTPUT
110 INPUT A$
120 IF A$="$END" THEN CLOSE #6:STOP
130 INPUT D,E
140 PRINT #6,A$,D,E,
150 GOTO 110
```

PRINT # and Data Formats

Refer to PRINT (with files) for information on how the PRINT statement writes a record in internal- or display-type data format. Note that if you print a file in display-type format that the computer will later read, 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 is read from the file, the computer separates the fields by the comma separators placed between them.

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

```
100 OPEN #6,"1.PENDING",DISPLAY,OUTPUT
110 INPUT A$
120 IF A$="$END" THEN CLOSE #6:STOP
130 INPUT D,E
140 PRINT #6,A$; ",";D; ","; E; ",";
150 GOTO 110
```

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 occurs. The CLOSE statement can be used to delete a file on some peripheral devices.

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

```
100 OPEN #3,"2.CFILE",INTERNAL
110 INPUT #3, A$,D,E
120 CLOSE #3
```

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 if 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 #3,"2.CFILE",INTERNAL
105 IF EOF(3) THEN CLOSE #3:STOP
110 INPUT #3, A$,D,E
115 PRINT A$,D,E:PAUSE 1
117 GOTO 105
```

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 #5, 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 1. The data is read from file #5 and printed on file #1. When the end-of-file is reached, the message End of data is displayed.

```
100 OPEN #5,"1.RESFILE",INTERNAL
110 INPUT "Enter street: ";ST$
120 INPUT "Name: ";A$
130 IF A$ = "END$" THEN 170
140 INPUT "Address: ";B$,"Zip: ";C$
150 PRINT #5,A$&"family",B$&" "&ST$,"794"&C$
160 GOTO 110
170 RESTORE #5
180 OPEN #1,"50",OUTPUT
190 IF EOF(5) THEN PRINT "End of data":PAUSE:CLOSE #5:STOP
200 INPUT #5,NAMES$,ST$,ZIP$
210 PRINT #1,NAMES$,ST$,ZIP$
220 GOTO 190
```