CHAPTER 2

BASIC LANGUAGE VOCABULARY

- Introduction
- BASIC Keywords, Abbreviations, and Function Types
- Description of BASIC Keywords (Alphabetical)
- The COMMODORE 64 Keyboard and Features
- Screen Editor

INTRODUCTION

This chapter explains *CBM BASIC Language keywords*. First we give you an easy to read list of keywords, their abbreviations and what each letter looks like on the screen. Then we explain how the syntax and operation of each keyword works in detail, and examples are shown to give you an idea as to how to use them in your programs.

As a convenience, Commodore 64 BASIC allows you to abbreviate most keywords. Abbreviations are entered by typing enough letters of the keyword to distinguish it from all other keywords, with the last letter or graphics entered holding down the SHIFT key.

Abbreviations do NOT save any memory when they're used in programs, because all keywords are reduced to single-character "tokens" by the BASIC Interpreter. When a program containing abbreviations is listed, all keywords appear in their fully spelled form. You can use abbreviations to put more statements onto a program line even if they won't fit onto the 80-character logical screen line. The Screen Editor works on an 80-character line. This means that if you use abbreviations on any line that goes over 80 characters, you will NOT be able to edit that line when LISTed. Instead, what you'll have to do is (1) retype the entire line including all abbreviations, or (2) break the single line of code into two lines, each with its own line number, etc.

A complete list of keywords, abbreviations, and their appearance on the screen is presented in Table 2-1. They are followed by an alphabetical description of all the statements, commands, and functions available on your Commodore 64.

This chapter also explains the BASIC functions built into the BASIC Language Interpreter. Built-in functions can be used in direct mode statements or in any program, without having to define the function further. This is NOT the case with user-defined functions. The results of built-in BASIC functions can be used as immediate output or they can be assigned to a variable name of an appropriate type. There are two types of BASIC functions:

1) NUMERIC
2) STRING

Arguments of built-in functions are always enclosed in parentheses ( ). The parentheses always come directly after the function keyword and NO SPACES between the last letter of the keyword and the left parenthesis (.

The type of argument needed is generally decided by the data type in the result. Functions which return a string value as their result are identified by having a dollar sign ($) as the last character of the keyword. In some cases string functions contain one or more numeric argument.

Numeric functions will convert between integer and floating-point format as needed. In the descriptions that follow, the data type of the value returned is shown with each function name. The types of arguments are also given with the statement format.

Table 2-1. COMMODORE 64 BASIC KEYWORDS

COMMAND	ABBREVIATION	SCREEN	FUNCTION TYPE
ABS	A SHIFT B	A	NUMERIC
AND	A SHIFT N	A	
ASC	A SHIFT S	A	NUMERIC
ATN	A SHIFT T	A	NUMERIC
CHR$	C SHIFT H	C	STRING
CLOSE	CL SHIFT O	CL	
CLR	C SHIFT L	C	
CMD	C SHIFT M	C	
CONT	C SHIFT O	C	
COS	none	COS	NUMERIC
DATA	D SHIFT A	D	
DEF	D SHIFT E	D	
DIM	D SHIFT I	D	

COMMAND	ABBREVIATION	SCREEN	FUNCTION TYPE
END	E SHIFT N	E	
EXP	E SHIFT X	E	NUMERIC
FN	none	FN	
FOR	F SHIFT O	F	
FRE	F SHIFT R	F	NUMERIC
GET	G SHIFT E	G	
GET#	none	GET#	
GOSUB	GO SHIFT S	GO	
GOTO	G SHIFT O	G	
IF	none	IF	
INPUT	none	INPUT	
INPUT#	I SHIFT N	I	
INT	none	INT	NUMERIC
LEFT$	LE SHIFT F	LE	STRING
LEN	none	LEN	NUMERIC
LET	L SHIFT E	L	
LIST	L SHIFT I	L	
LOAD	L SHIFT O	L	
LOG	none	LOG	NUMERIC

COMMAND	ABBREVIATION	SCREEN	FUNCTION TYPE
MID$	M SHIFT I	M	STRING
NEW	none	NEW	
NEXT	N SHIFT E	N	
NOT	N SHIFT O	N	
ON	none	ON	
OPEN	O SHIFT P	O	
OR	none	OR	
PEEK	P SHIFT E	P	NUMERIC
POKE	P SHIFT O	P	
POS	none	POS	NUMERIC
PRINT	?	?	
PRINT#	P SHIFT R	P	
READ	R SHIFT E	R	
REM	none	REM	
RESTORE	RE SHIFT S	RE	
RETURN	RE SHIFT T	RE	
RIGHT$	R SHIFT I	R	STRING
RND	R SHIFT N	R	NUMERIC
RUN	R SHIFT U	R	

COMMAND	ABBREVIATION	SCREEN	FUNCTION TYPE
SAVE	S SHIFT A	S 	
SGN	S SHIFT G	S 	NUMERIC
SIN	S SHIFT I	S 	NUMERIC
SPC(	S SHIFT P	S 	SPECIAL
SQR	S SHIFT Q	S 	NUMERIC
STATUS	ST	ST	NUMERIC
STEP	ST SHIFT E	ST 	
STOP	S SHIFT T	S 	
STR$	ST SHIFT R	ST 	STRING
SYS	S SHIFT Y	S 	
TAB(	T SHIFT A	T 	SPECIAL
TAN	none	TAN	NUMERIC
THEN	T SHIFT H	T 	
TIME	TI	TI	NUMERIC
TIME$	TI$	TI$	STRING
TO	none	TO	
USR	U SHIFT S	U 	NUMERIC
VAL	V SHIFT A	V 	NUMERIC
VERIFY	V SHIFT E	V 	
WAIT	W SHIFT A	W 	

DESCRIPTION OF BASIC KEYWORDS

ABS

TYPE: Function-Numeric
FORMAT: ABS(<expression>)

Action: Returns the absolute value of the number, which is its value without any signs. The absolute value of a negative number is that number multiplied by −1.

EXAMPLES of ABS Function:

```
10 X = ABS ( Y )
10 PRINT ABS ( X * J )
10 IF X = ABS (X) THEN PRINT "POSITIVE"
```

AND

TYPE: Operator
FORMAT: <expression> AND <expression>

Action: AND is used in Boolean operations to test bits. It is also used in operations to check the truth of both operands.

In Boolean algebra, the result of an AND operation is 1 only if both numbers being ANDed are 1. The result is 0 if either or both is 0 (false).

EXAMPLES of 1-Bit AND Operation:

0	1	0	1
AND 0	AND 0	AND 1	AND 1
0	0	0	1

The Commodore 64 performs the AND operation on numbers in the range from −32768 to −32767. Any fractional values are not used, and numbers beyond the range will cause an **?ILLEGAL QUANTITY** error

message. When converted to binary format, the range allowed yields 16 bits for each number. Corresponding bits are ANDed together, forming a 16-bit result in the same range.

EXAMPLES of 16-Bit AND Operation:

```
                    17
                AND 194
                -------
          0000000000010001
      AND 0000000011000010
-------------------------------
(BINARY) 0000000000000000
-------------------------------
(DECIMAL)                 0
```

```
                 32007
             AND 28761
             ---------
       0111110100000111
   AND 0111000001011001
---------------------------------
(BINARY) 0111000000000001
---------------------------------
(DECIMAL)          28673
```

```
                  —241
              AND 15359
              ---------
       1111111100001111
   AND 0011101111111111
---------------------------------
(BINARY) 0011101100001111
---------------------------------
(DECIMAL)          15119
```

When evaluating a number for truth or falsehood, the computer assumes the number is true as long as its value isn't 0. When evaluating a comparison, it assigns a value of −1 if the result is true, while false has a value of 0. In binary format, −1 is all 1's and 0 is all 0's. Therefore, when ANDing true/false evaluations, the result will be true if any bits in the result are true.

EXAMPLES of Using AND with True/False Evaluations:

```
50 IF X=7 AND W=3 THEN GOTO 10: REM ONLY TRUE IF BOTH X=7
   AND W=3 ARE TRUE
60 IF A AND Q=7 THEN GOTO 10: REM TRUE IF A IS NON-ZERO
   AND Q=7 IS TRUE
```

ASC

TYPE: Function-Numeric
FORMAT: ASC (<string>)

Action: ASC will return a number from 0 to 255 which corresponds to the Commodore ASCII value of the first character in the string. The table of Commodore ASCII values is shown in Appendix C.

EXAMPLES OF ASC Function:

```
10 PRINT ASC("Z")
20 X = ASC("ZEBRA")
30 J = ASC(J$)
```

If there are no characters in the string, an **?ILLEGAL QUANTITY** error results. In the third example above, if J$="", the ASC function will not work. The GET and GET# statement read a CHR$(0) as a null string. To eliminate this problem, you should add a CHR$(0) to the end of the string as shown below.

EXAMPLE of ASC Function Avoiding ILLEGAL QUANTITY ERROR:

```
30 J = ASC(J$ + CHR$(0))
```

ATN

TYPE: Function-Numeric
FORMAT: ATN (<number>)

Action: This mathematical function returns the arctangent of the number. The result is the angle (in radians) whose tangent is the number given. The result is always in the range $-\pi/2$ to $+\pi/2$.

EXAMPLES of ATN Function:

```
10 PRINT ATN ( 0 )
20 X = ATN ( J ) * 180 / π : REM CONVERT TO DEGREES
```

CHR$

TYPE: Function-String
FORMAT: CHR$ (<number>)

Action: This function converts a Commodore ASCII code to its character equivalent. See Appendix C for a list of characters and their codes. The number must have a value between 0 and 255, or an **?ILLEGAL QUANTITY** error message results.

EXAMPLES of CHR$ Function:

```
10 PRINT CHR$(65) : REM 65 = UPPER CASE A
20 A$ = CHR$(13) : REM 13 = RETURN KEY
50 A = ASC(A$) : A$ = CHR$(A): REM CONVERTS TO C64 ASCII
   CODE AND BACK
```

CLOSE

TYPE: I/O Statement
FORMAT: CLOSE <file number>

Action: This statement shuts off any data file or channel to a device. The file number is the same as when the file or device was OPENed (see OPEN statement and the section on INPUT/OUTPUT programming).

When working with storage devices like cassette tape and disks, the CLOSE operation stores any incomplete buffers to the device. When this is not performed, the file will be incomplete on the tape and unreadable on the disk. The CLOSE operation isn't as necessary with other devices, but it does free up memory for other files. See your external device manual for more details.

EXAMPLES of CLOSE Statement:

```
10 CLOSE 1
20 CLOSE X
30 CLOSE 9 * ( 1 + J )
```

CLR

TYPE: Statement
FORMAT: CLR

Action: This statement makes available RAM memory that had been used but is no longer needed. Any BASIC program in memory is untouched, but all variables, arrays, GOSUB addresses, FOR. . . NEXT loops, user-defined functions, and files are erased from memory, and their space is made available to new variables, etc.

In the case of files to the disk and cassette tape, they are *not* properly CLOSEd by the CLR statement. The information about the files is lost to the computer, including any incomplete buffers. The disk drive will still think the file is OPEN. See the CLOSE statement for more information on this.

EXAMPLE of CLR Statement:

```
10 X=25
20 CLR
30 PRINT X

RUN
0

READY
```

CMD

TYPE: I/O Statement
FORMAT: CMD <file number> [, string]

Action: This statement switches the primary output device from the TV screen to the file specified. This file could be on disk, tape, printer, or an I/O device like the modem. The file number must be specified in a prior OPEN statement. The string, when specified, is sent to the file. This is handy for titling printouts, etc.

When this command is in effect, any PRINT statements and LIST commands will not display on the screen, but will send the text in the same format to the file.

To re-direct the output back to the screen, the PRINT# command should send a blank line to the CMD device before CLOSEing, so it will stop expecting data (called "un-listening" the device).

Any system error (like **?SYNTAX ERROR**) will cause output to return to the screen. Devices aren't un-listened by this, so you should send a blank line after an error condition. (See your printer or disk manual for more details.)

EXAMPLES of CMD Statement:

```
OPEN 4, 4: CMD 4, "TITLE" : LIST: REM LISTS PROGRAM ON PRINTER
PRINT# 4: CLOSE 4: REM UN-LISTENS AND CLOSES PRINTER

10 OPEN 1, 1, 1, "TEST": REM CREATE SEQ FILE
20 CMD 1: REM OUTPUT TO TAPE FILE, NOT SCREEN
30 FOR L = 1 TO 100
40 PRINT L: REM PUTS NUMBER IN TAPE BUFFER
50 NEXT
60 PRINT# 1: REM UNLISTEN
70 CLOSE 1: REM WRITE UNFINISHED BUFFER, PROPERLY FINISH
```

CONT

TYPE: Command
FORMAT: CONT

Action: This command re-starts the execution of a program which was halted by a STOP or END statement or the RUN/STOP key being pressed. The program will re-start at the exact place from which it left off.

While the program is stopped, the user can inspect or change any variables or look at the program. When de-bugging or examining a program, STOP statements can be placed at strategic locations to allow examination of variables and to check the flow of the program.

The error message **CAN'T CONTINUE** will result from editing the program (even just hitting RETURN with the cursor on an unchanged line), or if the program halted due to an error, or if you caused an error before typing CONT to re-start the program.

EXAMPLE of CONT Command:

```
10 PI=0:C=1
20 PI=PI+4/C-4/(C+2)
30 PRINT PI
40 C=C+4:GOTO 20
```

This program calculates the value of PI. RUN this program, and after a short while hit the RUN/STOP key. You will see the display:

BREAK IN 20 | NOTE: Might be different number.

Type the command PRINT C to see how far the Commodore 64 has gotten. Then use CONT to resume from where the Commodore 64 left off.

COS

TYPE: Function
FORMAT: COS (<number>)

Action: This mathematical function calculates the cosine of the number, where the number is an angle in radians.

EXAMPLES of COS Function:

```
10 PRINT COS ( 0 )
20 X = COS ( Y * π / 180 ) : REM CONVERT DEGREES TO RADIANS
```

DATA

TYPE: Statement
FORMAT: DATA <list of constants>

Action: DATA statements store information within a program. The program uses the information by means of the READ statement, which pulls successive constants from the DATA statements.

The DATA statements don't have to be executed by the program, they only have to be present. Therefore, they are usually placed at the end of the program.

All data statements in a program are treated as a continuous list. Data is READ from left to right, from the lowest numbered line to the highest. If the READ statement encounters data that doesn't fit the type requested (if it needs a number and finds a string) an error message occurs.

Any characters can be included as data, but if certain ones are used the data item must be enclosed by quote marks (" "). These include punctuation like comma (,), colon (:), blank spaces, and shifted letters, graphics, and cursor control characters.

EXAMPLES of DATA Statement:

```
10 DATA 1, 10, 5, 8
20 DATA JOHN, PAUL, GEORGE, RINGO
30 DATA "DEAR MARY, HOW ARE YOU, LOVE, BILL"
40 DATA -1.7E-9, 3.33
```

DEF FN

TYPE: Statement
FORMAT: DEF FN <name> (<variable>) = <expression>

Action: This sets up a user-defined function that can be used later in the program. The function can consist of any mathematical formula. Usec-defined functions save space in programs where a long formula is used in several places. The formula need only be specified once, in the definition statement, and then it is abbreviated as a function name. It must be executed once, but any subsequent executions are ignored.

The function name is the letters FN followed by any variable name. This can be 1 or 2 characters, the first being a letter and the second a letter or digit.

EXAMPLES of DEF FN Statement:

```
10 DEF FN A (X) = X + 7
20 DEF FN AA (X) = Y * Z
30 DEF FNA9 (Q) = INT( RND( 1)* Q+ 1)
```

The function is called later in the program by using the function name with a variable in parentheses. This function name is used like any other variable, and its value is automatically calculated.

EXAMPLES of FN Use:

```
40 PRINT FN A (9)
50 R = FNAA (9)
60 G = G + FN A9 (10)
```

In line 50 above, the number 9 inside the parentheses does not affect the outcome of the function, because the function definition in line 20 doesn't use the variable in the parentheses. The result is Y times Z, regardless of the value of X. In the other two functions, the value in parentheses does affect the result.

DIM

TYPE: Statement
FORMAT: DIM <variable> (<subscripts>) [, <variable> (<subscripts>) . . .]

Action: This statement defines an array or matrix of variables. This allows you to use the variable name with a subscript. The subscript points to the element being used. The lowest element number in an array is zero, and the highest is the number given in the DIM statement, which has a maximum of 32767.

The DIM statement must be executed once *and only* once for each array. A **REDIM'D ARRAY** error occurs if this line is re-executed. Therefore, most programs perform all DIM operations at the very beginning.

There may be any number of dimensions and 255 subscripts in an array, limited only by the amount of RAM memory which is available to hold the variables. The array may be made up of normal numeric variables, as shown above, or of strings or integer numbers. If the variables are other than normal numeric, use the $ or % signs after the variable name to indicate string or integer variables,

If an array referenced in a program was never DIMensioned, it is automatically dimensioned to 11 elements in each dimension used in the first reference.

EXAMPLES of DIM Statement:

```
10 DIM A ( 100 )
20 DIM Z ( 5, 7), Y ( 3, 4, 5)
30 DIM Y7% ( Q )
40 DIM PH$ (1000)
50 F (4) =9: REM AUTOMATICALLY PERFORMS DIM F (10)
```

EXAMPLE of FOOTBALL SCORE-KEEPING Using DIM:

```
10 DIM S(1,5), T$(1)
20 INPUT "TEAM NAMES"; T$(0), T$(1)
30 FOR Q=1 TO 5: FOR T=0 TO 1
40 PRINT T$(T), "SCORE IN QUARTER" Q
50 INPUT S(T,Q): S(T,0)= S(T,0)+ S(T,Q)
60 NEXT T,Q
70 PRINT CHR$(147) "SCOREBOARD"
80 PRINT "QUARTER"
90 FOR Q=1 TO 5
100 PRINT TAB( Q*2 +9) Q;
110 NEXT: PRINT TAB(15) "TOTAL"
120 FOR T=0 TO 1: PRINT T$(T);
130 FOR Q=1 TO 5
140 PRINT TAB(Q*2 +9) S(T,Q);
150 NEXT: PRINT TAB(15) S(T,0)
160 NEXT
```

CALCULATING MEMORY USED BY DIM:

5 bytes for the array name
2 bytes for each dimension
2 bytes/element for integer variables
5 bytes/element for normal numeric variables
3 bytes/element for string variables
1 byte for each character in each string element

END

TYPE: Statement
FORMAT: END

Action: This finishes a program's execution and displays the READY message, returning control to the person operating the computer. There may be any number of END statements within a program. While it is not necessary to include any END statements at all, it is recommended that a program does conclude with one, rather than just running out of lines.

The END statement is similar to the STOP statement. The only difference is that STOP causes the computer to display the message **BREAK IN LINE XX** and END just displays READY. Both statements allow the computer to resume execution by typing the CONT command.

EXAMPLES of END Statement:

```
10 PRINT "DO YOU REALLY WANT TO RUN THIS PROGRAM"
20 INPUT A$
30 IF A$ = "NO" THEN END
40 REM REST OF PROGRAM . . .
999 END
```

EXP

TYPE: Function-Numeric
FORMAT: EXP (<number>)

Action: This mathematical function calculates the constant e (2.71828183) raised to the power of the number given. A value greater than 88.0296919 causes an ?OVERFLOW error to occur.

EXAMPLES of EXP Function:

```
10 PRINT EXP (1)
20 X = Y * EXP (Z * Q)
```

FN

TYPE: Function-Numeric
FORMAT: FN <name> (<number>)

Action: This function references the previously DEFined formula specified by name. The number is substituted into its place (if any) and the formula is calculated. The result will be a numeric value.

This function can be used in direct mode, as long as the statement DEFining it has been executed.

If an FN is executed before the DEF statement which defines it, an **UNDEF'D FUNCTION** error occurs.

EXAMPLES of FN (User-Defined) Function:

```
PRINT FN A ( Q )
1100 J = FN J (7) + FN J (9)
9990 IF FN B7 (I+1)= 6 THEN END
```

FOR . . . TO . . . [STEP . . .]

TYPE: Statement
FORMAT: FOR <variable> = <start> TO <limit> [STEP <increment>]

Action: This is a special BASIC statement that lets you easily use a variable as a counter. You must specify certain parameters: the floating-point variable name, its starting value, the limit of the count, and how much to add during each cycle.

Here is a simple BASIC program that counts from 1 to 10, PRINTing each number and ENDing when complete, and using no FOR statements:

```
100 L = 1
110 PRINT L
120 L = L + 1
130 IF L <= 10 THEN 110
140 END
```

Using the FOR statement, here is the same program:

```
100 FOR L = 1 TO 10
110 PRINT L
120 NEXT L
130 END
```

As you can see, the program is shorter and easier to understand using the FOR statement.

When the FOR statement is executed, several operations take place. The <start> value is placed in the <variable> being used in the counter. In the example above, a 1 is placed in L.

When the NEXT statement is reached, the <increment> value is added to the <variable>. If a STEP was not included, the <increment> is set to +1. The first time the program above hits line 120, 1 is added to L, so the new value of L is 2.

Now the value in the <variable> is compared to the <limit>. If the <limit> has not been reached yet, the program GOes TO the line after the original FOR statement. In this case, the value of 2 in L is less than the limit of 10, so it GOes TO line 110.

Eventually, the value of <limit> is exceeded by the <variable>. At that time, the loop is concluded and the program continues with the line following the NEXT statement. In our example, the value of L reaches 11, which exceeds the limit of 10, and the program goes on with line 130.

When the value of <increment> is positive, the <variable> must exceed the <limit>, and when it is negative it must become less than the <limit>.

> **NOTE:** A loop always executes at least once.

EXAMPLES of FOR. . .TO. . .STEP. . .Statement:

```
100 FOR L = 100 TO 0 STEP -1
100 FOR L = PI TO 6*π STEP .01
100 FOR AA = 3 TO 3
```

FRE

TYPE: Function
FORMAT: FRE (<variable>)

Action: This function tells you how much RAM is available for your program and its variables. If a program tries to use more space than is available, the **OUT OF MEMORY** error results.

The number in parentheses can have any value, and it is not used in the calculation.

> **NOTE:** If the result of FRE is negative, add 65536 to the FRE number to get the number of bytes available in memory.

EXAMPLES of FRE Function:

```
PRINT FRE ( 0 )
10 X = ( FRE ( K ) - 1000 ) / 7
950 IF FRE ( 0 ) < 100 THEN PRINT "NOT ENOUGH ROOM"
```

> **NOTE:** The following always tells you the current available RAM:
>
> ```
> PRINT FRE(0) - (FRE(0) < 0)* 65536
> ```

GET

TYPE: Statement
FORMAT: GET <variable list>

Action: This statement reads each key typed by the user. As the user is typing, the characters are stored in the Commodore 64's keyboard buffer. Up to 10 characters are stored here, and any keys struck after the 10th are lost. Reading one of the characters with the GET statement makes room for another character.

If the GET statement specifies numeric data, and the user types a key other than a number, the message **?SYNTAX ERROR** appears. To be safe, read the keys as strings and convert them to numbers later.

The GET statement can be used to avoid some of the limitations of the INPUT statement. For more on this, see the section on Using the GET Statement in the Programming Techniques section.

EXAMPLES of GET Statement:

```
10 GET A$: IF A$ = "" THEN 10: REM LOOPS IN 10 UNTIL ANY KEY
   HIT
20 GET A$, B$, C$, D$, E$: REM READS 5 KEYS
30 GET A, A$
```

GET#

TYPE: I/O Statement
FORMAT: GET# <file number>, <variable list>

Action: This statement reads characters one-at-a-time from the device or file specified. It works the same as the GET statement, except that the data comes from a different place than the keyboard. If no character is received, the variable is set to an empty string (equal to "") or to 0 for numeric variables. Characters used to separate data in files, like the comma (,) or RETURN key code (ASC code of 13), are received like any other character.

When used with device #3 (TV screen), this statement will read characters one by one from the screen. Each use of GET# moves the cursor 1 position to the right. The character at the end of the logical line is changed to a CHR$ (13), the RETURN key code.

EXAMPLES of GET# Statement:

```
5 GET# 1, A$
10 OPEN 1, 3: GET# 1, Z7$
20 GET# 1, A, B, C$, D$
```

GOSUB

TYPE: Statement
FORMAT: GOSUB <line number>

Action: This is a specialized form of the GOTO statement, with one important difference: GOSUB remembers where it came from. When the RETURN statement (different from the RETURN key on the keyboard) is reached in the program, the program jumps back to the statement immediately following the original GOSUB statement.

The major use of a subroutine (GOSUB really means GO to a SUBroutine) is when a small section of program is used by different sections of the program. By using subroutines rather than repeating the same lines over and over at different places in the program, you can save lots of program space. In this way, GOSUB is similar to DEF FN. DEF FN lets you save space when using a formula, while GOSUB saves space when using a several-line routine. Here is an inefficient program that doesn't use GOSUB:

```
100 PRINT "THIS PROGRAM PRINTS"
110 FOR L = 1 TO 500 : NEXT
120 PRINT "SLOWLY ON THE SCREEN"
130 FOR L = 1 TO 500 : NEXT
140 PRINT "USING A SIMPLE LOOP"
150 FOR L = 1 TO 500 : NEXT
160 PRINT "AS A TIME DELAY."
170 FOR L = 1 TO 500 : NEXT
```

Here is the same program using GOSUB:

```
100 PRINT "THIS PROGRAM PRINTS"
110 GOSUB 200
120 PRINT "SLOWLY ON THE SCREEN"
130 GOSUB 200
140 PRINT "USING A SIMPLE LOOP"
150 GOSUB 200
160 PRINT "AS A TIME DELAY."
170 GOSUB 200
180 END
200 FOR L = 1 TO 500 : NEXT
210 RETURN
```

Each time the program executes a GOSUB, the line number and position in the program line are saved in a special area called the "stack," which takes up 256 bytes of your memory. This limits the amount of data that can be stored in the stack. Therefore, the number of subroutine return addresses that can be stored is limited, and care should be taken to make sure every GOSUB hits the corresponding RETURN, or else you'll run out of memory even though you have plenty of bytes free.

GOTO

TYPE: Statement
FORMAT: GOTO <line number>
or GO TO <line number>

Action: This statement allows the BASIC program to execute lines out of numerical order. The word GOTO followed by a number will make the program jump to the line with that number. GOTO NOT followed by a number equals GOTO 0. It must have the line number after the word GOTO.

It is possible to create loops with GOTO that will never end. The simplest example of this is a line that GOes TO itself, like 10 GOTO 10. These loops can be stopped using the RUN/STOP key on the keyboard.

EXAMPLES of GOTO Statement:

```
GOTO 100
10 GO TO 50
20 GOTO 999
```

IF . . . THEN . . .

TYPE: Statement
FORMAT: IF <expression> THEN <line number>
IF <expression> GOTO <line number>
IF <expression> THEN <statements>

Action: This is the statement that gives BASIC most of its "intelligence," the ability to evaluate conditions and take different actions depending on the outcome.

The word IF is followed by an expression, which can include variables, strings, numbers, comparisons, and logical operators. The word THEN appears *on the same line* and is followed by either a line number or one or more BASIC statements. When the expression is false, everything after the word THEN on that line is ignored, and execution continues with the next line number in the program. A true result makes the program either branch to the line number after the word THEN or execute whatever other BASIC statements are found on that line.

EXAMPLE of IF. . .GOTO. . .Statement:

```
100 INPUT "TYPE A NUMBER"; N
110 IF N <= 0 GOTO 200
120 PRINT "SQUARE ROOT=" SQR(N)
130 GOTO 100
200 PRINT "NUMBER MUST BE >0"
210 GOTO 100
```

This program prints out the square root of any positive number. The IF statement here is used to validate the result of the INPUT. When the result of N <= 0 is true, the program skips to line 200, and when the result is false the next line to be executed is 120. Note that THEN GOTO is not needed with IF. . .THEN, as in line 110 where GOTO 200 actually means THEN GOTO 200.

EXAMPLE OF IF. . . THEN. . . Statement:

```
100 FOR L = 1 TO 100
110 IF RND(1) < .5 THEN X = X+ 1 : GOTO 130
120 Y = Y + 1
130 NEXT L
140 PRINT "HEADS= " X
150 PRINT "TAILS= " Y
```

The IF in line 110 tests a random number to see if it is less than .5. When the result is true, the whole series of statements following the word THEN are executed: first X is incremented by 1, then the program skips to line 130. When the result is false, the program drops to the next statement, line 120.

INPUT

TYPE: Statement
FORMAT: INPUT ["<prompt>" ;] <variable list>

Action: This is a statement that lets the person RUNning the program "feed" information into the computer. When executed, this statement PRINTs a question mark (?) on the screen, and positions the cursor 1 space to the right of the question mark. Now the computer waits, cursor blinking, for the operator to type in the answer and press the RETURN key.

The word INPUT may be followed by any text contained in quote marks (" "). This text is PRINTed on the screen, followed by the question mark.

After the text comes a semicolon (;) and the name of one or more variables separated by commas. This variable is where the computer stores the information that the operator types. The variable can be any legal variable name, and you can have several different variable names, each for a different input.

EXAMPLES of INPUT Statement:

```
100 INPUT A
110 INPUT B, C, D
120 INPUT "PROMPT"; E
```

When this program RUNs,the question mark appears to prompt the operator that the Commodore 64 is expecting an input for line 100. Any number typed in goes into A, for later use in the program. If the answer typed was not a number, the **?REDO FROM START** message appears, which means that a string was received when a number was expected. If the operator just hits RETURN without typing anything, the variable's value doesn't change.

Now the next question mark, for line 110, appears. If we type only one number and hit RETURN , the Commodore 64 will now display 2 question marks (??), which means that more input is required. You can

just type as many inputs as you need separated by commas, which prevents the double question mark from appearing. If you type more data than the INPUT statement requested, the **?EXTRA IGNORED** message appears, which means that the extra items you typed were not put into any variables.

Line 120 displays the word PROMPT before the question mark appears. The semicolon is required between the prompt and any list of variables.

The INPUT statement can never be used outside a program. The Commodore 64 needs space for a buffer for the INPUT variables, the same space that is used for commands.

INPUT#

TYPE: I/O Statement
FORMAT: INPUT# <file number> , <variable list>

Action: This is usually the fastest and easiest way to retrieve data stored in a file on disk or tape. The data is in the form of whole variables of up to 80 characters in length, as opposed to the one-at-a-time method of GET#. First, the file must have been OPENed, then INPUT# can fill the variables.

The INPUT# command assumes a variable is finished when it reads a RETURN code (CHR$ (13)), a comma (,), semicolon (;), or colon (:). Quote marks can be used to enclose these characters when writing if they are needed (see PRINT# statement).

If the variable type used is numeric, and non-numeric characters are received, a **BAD DATA** error results. INPUT# can read strings up to 80 characters long, beyond which a **STRING TOO LONG** error results.

When used with device #3 (the screen), this statement will read an entire logical line and move the cursor down to the next line.

EXAMPLES of INPUT# Statement:

```
10 INPUT# 1, A
20 INPUT# 2, A$, B$
```

INT

TYPE: Integer Function
FORMAT: INT (<numeric>)

Action: Returns the integer value of the expression. If the expression is positive, the fractional part is left off. If the expression is negative, any fraction causes the next lower integer to be returned.

EXAMPLES of INT Function:

```
120 PRINT INT(99.4343), INT(-12.34)

 99          -13
```

LEFT$

TYPE: String Function
FORMAT: LEFT$ (<string>, <integer>)

Action: Returns a string comprised of the leftmost <integer> characters of the <string>. The integer argument value must be in the range 0 to 255. If the integer is greater than the length of the string, the entire string will be returned. If an <integer> value of zero is used, then a null string (of zero length) is returned.

EXAMPLES of LEFT$ Function:

```
10 A$ = "COMMODORE COMPUTERS"
20 B$ = LEFT$(A$,9): PRINT B$
RUN

COMMODORE
```

LEN

TYPE: Integer Function
Format: LEN (<string>)

Action: Returns the number of characters in the string expression. Non-printed characters and blanks are counted.

EXAMPLE of LEN Function:

```
CC$ = "COMMODORE COMPUTER": PRINT LEN(CC$)
 18
```

LET

TYPE: Statement
FORMAT: [LET] <variable> = <expression>

Action: The LET statement can be used to assign a value to a variable. But the word LET is optional and therefore most advanced programmers leave LET out because it's always understood and wastes valuable memory. The equal sign (=) alone is sufficient when assigning the value of an expression to a variable name.

EXAMPLES of LET Statement:

```
10 LET D= 12              (This is the same as D = 12)
20 LET E$ = "ABC"
30 F$ = "WORDS"
40 SUM$ = E$ + F$         (SUM$ would equal ABCWORDS)
```

LIST

TYPE: Command
FORMAT: LIST [[<first-line>]-[<last-line>]]

Action: The LIST command allows you to look at lines of the BASIC program currently in the memory of your Commodore 64. This lets you use your computer's powerful screen editor to edit programs which you've LISTed both quickly and easily.

The LIST system command displays all or part of the program that is currently in memory on the default output device. The LIST will normally be directed to the screen and the CMD statement can be used to switch output to an external device such as a printer or a disk. The LIST command can appear in a program, but BASIC always returns to the system READY message after a LIST is executed.

When you bring the program LIST onto the screen, the "scrolling" of the display from the bottom of the screen to the top can be slowed by holding down the ConTRoL CTRL key. LIST is aborted by typing the RUN/STOP key.

If no line-numbers are given the entire program is listed. If only the first-line number is specified, and followed by a hyphen (-), that line and all higher-numbered lines are listed. If only the last line-number is specified, and it is preceded by a hyphen, then all lines from the beginning of the program through that line are listed. If both numbers are specified, the entire range, including the line-numbers LISTed, is displayed.

EXAMPLES of LIST Command:

```
LIST            (Lists the program currently in memory.)

LIST 500        (Lists line 500 only.)

LIST 150-       (Lists all lines from 150 to the end.)

LIST -1000      (Lists all lines from the lowest through 1000.)

LIST 150-1000   (Lists lines 150 through 1000, inclusive.)

10 PRINT "THIS IS LINE 10"
20 LIST           (LIST used in Program Mode)
30 PRINT "THIS IS LINE 30"
```

LOAD

TYPE: Command
FORMAT: LOAD ["<file-name>"] [,<device>] [,<address>]

Action: The LOAD statement reads the contents of a program file from tape or disk into memory. That way you can use the information LOADed or change the information in some way. The device number is optional, but when it is left out the computer will automatically default to 1, the cassette unit. The disk unit is normally device number 8. The LOAD closes all open files and, if it is used in direct mode, it performs a CLR (clear) before reading the program. If LOAD is executed from within a program, the program is RUN. This means that you can use LOAD to "chain" several programs together. None of the variables are cleared during a chain operation.

If you are using file-name pattern matching, the first file which matches the pattern is loaded. The asterisk in quotes by itself ("*") causes the first file-name in the disk directory to be loaded. If the file-name used does not exist or if it is not a program file, the BASIC error message **?FILE NOT FOUND** occurs.

When LOADing programs from tape, the <file-name> can be left out, and the next program file on the tape will be read. The Commodore 64 will blank the screen to the border color after the PLAY key is pressed. When the program is found, the screen clears to the background color and the "FOUND" message is displayed. When the C= key, CTRL key, ← key, or SPACE BAR is pressed, the file will be loaded. Programs will LOAD starting at memory location 2048 unless a secondary <address> of 1 is used. If you use the secondary address of 1 this will cause the program to LOAD to the memory location from which it was saved.

EXAMPLES of LOAD Command:

```
LOAD
```
(Reads the next program on tape)

```
LOAD A$
```
(Uses the name in A$ to search)

```
LOAD "*",8
```
(LOADs first program from disk)

```
LOAD "",1,1
```
(Looks for the first program on tape, and LOADs it into the same part of memory that it came from)

```
LOAD "STAR TREK"
PRESS PLAY ON TAPE
FOUND STAR TREK
LOADING
READY.
```
(LOAD a file from tape)

```
LOAD "FUN",8
SEARCHING FOR FUN
LOADING
READY.
```
(LOAD a file from disk)

```
LOAD "GAME ONE",8,1
SEARCHING FOR GAME ONE
LOADING
READY.
```
(LOAD a file to the specific memory location from which the program was saved on the disk)

LOG

TYPE: Floating-Point Function
FORMAT: LOG (<numeric>)

Action: Returns the natural logarithm (log to the base of e) of the argument. If the value of the argument is zero or negative the BASIC error message **?ILLEGAL QUANTITY** will occur.

EXAMPLES of LOG Function:

```
25 PRINT LOG(45/7)
 1.86075234

10 NUM = LOG(ARG) / LOG(10) (Calculates the LOG of ARG to the
                             base 10)
```

MID$

TYPE: String Function
FORMAT: MID$ (<string>, <numeric-1> [,<numeric-2>])

Action: The MID$ function returns a sub-string which is taken from within a larger <string> argument. The starting position of the sub-string is defined by the <numeric-1> argument and the length of the sub-string by the <numeric-2> argument. Both of the numeric arguments can have values ranging from 0 to 255.

If the <numeric-1> value is greater than the length of the <string>, or if the <numeric-2> value is zero, then MID$ gives a null string value. If the <numeric-2> argument is left out, then the computer will assume that a length of the rest of the string is to be used. And if the source string has fewer characters than <numeric-2>, from the starting position to the end of the string argument, then the whole rest of the string is used.

EXAMPLE of MID$ Function:

```
10 A$="GOOD"
20 B$="MORNING EVENING AFTERNOON"
30 PRINT A$ — MID$(B$, 8, 8)

GOOD EVENING
```

NEW

TYPE: Command
FORMAT: NEW

Action: The NEW command is used to delete the program currently in memory and clear all variables. Before typing in a new program, NEW should be used in direct mode to clear memory. NEW can also be used in a program, but you should be aware of the fact that it will erase everything that has gone before and is still in the computer's memory. This can be particularly troublesome when you're trying to debug your program.

BE CAREFUL: Not clearing out an old program before typing a new one can result in a confusing mix of the two programs.

EXAMPLES of NEW Command:

NEW	(Clears the program and all variables)
10 NEW	(Performs a NEW operation and STOPs the program.)

NEXT

TYPE: Statement
FORMAT: NEXT [<counter>] [,<counter>] . . . .

Action: The NEXT statement is used with FOR to establish the end of a FOR. . . NEXT loop. The NEXT need not be physically the last statement in the loop, but it is always the last statement executed in a loop. The <counter> is the loop index's variable name used with FOR to start the loop. A single NEXT can stop several nested loops when it is followed by each FOR's <counter> variable name(s). To do this each name must appear in the order of inner-most nested loop first, to outer-most nested loop last. When using a single NEXT to increment and stop several variable names, each variable name must be separated by commas. Loops can be nested to 9 levels. If the counter variable(s) are omitted, the counter associated with the FOR of the current level (of the nested loops) is incremented.

When the NEXT is reached, the counter value is incremented by 1 or by an optional STEP value. It is then tested against an end-value to see if it's time to stop the loop. A loop will be stopped when a NEXT is found which has its counter value greater than the end-value.

EXAMPLES of NEXT Statement:

```
10 FOR J=1 TO 5: FOR K = 10 TO 20: FOR N = 5 TO -5 STEP -1

20 NEXT N, K, J                    (Stopping Nested Loops)

10 FOR L = 1 TO 100
20 FOR M = 1 TO 10
30 NEXT M
400 NEXT L                         (Note how the loops do NOT cross each
                                    other)

10 FOR A = 1 TO 10
20 FOR B = 1 TO 20
30 NEXT
40 NEXT                            (Notice that no variable names are
                                    needed)
```

NOT

TYPE: Logical Operator
FORMAT: NOT <expression>

Action: The NOT logical operator "complements" the value of each bit in its single operand, producing an integer "twos-complement" result. In other words, the NOT is really saying, "if it isn't. . . ". When working with a floating-point number, the operands are converted to integers and any fractions are lost. The NOT operator can also be used in a comparison to reverse the true/false value which was the result of a relationship test and therefore it will reverse the meaning of the comparison. In the first example below, if the "twos-complement" of "AA" is equal to "BB" and if "BB" is NOT equal to "CC" then the expression is true.

EXAMPLES of NOT Operator:

```
10 IF NOT AA = BB AND NOT(BB = CC) THEN . . . .

NN% = NOT 96: PRINT NN%
-97
```

> **NOTE:** *To find the value of NOT use the expression X=(−(X+1)). (The two's complement of any integer is the bit complement plus one.)*

ON

TYPE: Statement
FORMAT: ON <variable> GOTO / GOSUB <line-number> [,<line-number>] . . . .

Action: The ON statement is used to GOTO one of several given line-numbers, depending upon the value of a variable. The value of the variables can range from zero through the number of lines given. If the value is a non-integer, the fractional portion is left off. For example, if the variable value is 3, ON will GOTO the third line-number in the list.

If the value of the variable is negative, the BASIC error message **?ILLEGAL QUANTITY** occurs. If the number is zero, or greater than the number of items in the list, the program just "ignores" the statement and continues with the statement following the ON statement.

ON is really an underused variant of the IF. . .THEN. . . statement. Instead of using a whole lot of IF statements each of which sends the program to 1 specific line, 1 ON statement can replace a list of IF statements. When you look at the first example you should notice that the 1 ON statement replaces 4 IF. . .THEN. . . statements.

EXAMPLES of ON Statement:

```
ON −(A=7)−2*(A=3)− 3*(A<3)−4*(A>7)GOTO 400,900,1000,100

ON X GOTO 100,130,180,220

ON X+3 GOSUB 9000,20,9000

100 ON NUM GOTO 150, 300, 320, 390

500 ON SUM / 2 + 1 GOSUB 50, 80, 20
```

OPEN

TYPE: I/O Statement
FORMAT: OPEN <file-num>, [<device>] [,<address>] [,"<file-name> [,<type>] [,<mode>]"]

Action: This statement OPENs a channel for input and/or output to a peripheral device. However, you may NOT need all those parts for every OPEN statement. Some OPEN statements require only 2 codes:

1) LOGICAL FILE NUMBER
2) DEVICE NUMBER

The <file-num> is the logical file number, which relates the OPEN, CLOSE, CMD, GET#, INPUT#, and PRINT# statements to each other and associates them with the file-name and the piece of equipment being used. The logical file number can range from 1 to 255 and you can assign it any number you want in that range.

> **NOTE:** File numbers over 128 were really designed for other uses so it's good practice to use only numbers below 127 for file numbers.

Each peripheral device (printer, disk drive, cassette) in the system has its own number which it answers to. The <device> number is used with OPEN to specify on which device the data file exists. Peripherals like cassette decks, disk drives or printers also answer to several secondary addresses. Think of these as codes which tell each device what operation to perform. The device logical file number is used with every GET#, INPUT#, and PRINT#.

If the <device> number is left out the computer will automatically assume that you want your information to be sent to and received from the Datassette™, which is device number 1. The file-name can also be left out, but later on in your program, you can NOT call the file by name if you have not already given it one. When you are storing files on cassette tape, the computer will assume that the secondary <address> is zero (0) if you omit the secondary address (a READ operation).

A secondary address value of one (1) OPENs cassette tape files for writing. A secondary address value of two (2) causes an end-of-tape marker to be written when the file is later closed. The end-of-tape marker prevents accidentally reading past the end of data which results in the BASIC error message **?DEVICE NOT PRESENT.**

For disk files, the secondary addresses 2 thru 14 are available for data-files, but other numbers have special meanings in DOS commands. You must use a secondary address when using your disk drive(s). (See your disk drive manual for DOS command details.)

The <file-name> is a string of 1–16 characters and is optional for cassette or printer files. If the file <type> is left out the type of file will automatically default to the *Program* file unless the <mode> is given. *Sequential* files are OPENed for reading <mode>=R unless you specify that files should be OPENed for writing <mode>=W is specified. A file <type> can be used to OPEN an *existing* Relative file. Use REL for <type> with Relative files. Relative and Sequential files are for disk only.

If you try to access a file before it is OPENed the BASIC error message **?FILE NOT OPEN** will occur. If you try to OPEN a file for reading which does not exist the BASIC error message **?FILE NOT FOUND** will occur. If a file is OPENed to disk for writing and the file-name already exists, the DOS error message **FILE EXISTS** occurs. There is no check of this type available for tape files, so be sure that the tape is properly positioned or you might accidentally write over some data that had previously been SAVEd. If a file is OPENed that is already OPEN, the BASIC error message **FILE OPEN** occurs. (See Printer Manual for further details.)

EXAMPLES of OPEN Statements:

```
10 OPEN 2, 8, 4 "DISK-OUTPUT,
   SEQ,W"                          (Opens sequential file on disk)

10 OPEN 1, 1, 2, "TAPE-WRITE"      (Write End-of-File on Close)

10 OPEN 50, 0                      (Keyboard input)

10 OPEN 12, 3                      (Screen output)

10 OPEN 130, 4                     (Printer output)

10 OPEN 1,1,0, "NAME"              (Read from cassette)

10 OPEN 1,1,1, "NAME"              (Write to cassette)

10 OPEN 1,2,0, CHR$ (10)           (Open channel to RS-232 device)

10 OPEN 1,4,0, "STRING"            (Send upper case/graphics to
                                    the printer)
10 OPEN 1,4,7, "STRING"            (Send upper/lower case to
                                    printer)
10 OPEN 1,5,7, "STRING"            (Send upper/lower case to
                                    printer with device # 5)
10 OPEN 1,8,15, "COMMAND"          (Send a command to disk)
```

OR

TYPE: Logical Operator
FORMAT: <operand> OR <operand>

Action: Just as the relational operators can be used to make decisions egarding program flow, logical operators can connect two or more relations and return a true or false value which can then be used in a decision. When used in calculations, the logical OR gives you a bit result of 1 if the corresponding bit of either or both operands is 1. This will produce an integer as a result depending on the values of the operands. When used in comparisons the logical OR operator is also used to link two expressions into a single compound expression. If either of the expressions are true, the combined expression value is true (−1). In the first example below if AA is equal to BB OR if XX is 20, the expression is true.

Logical operators work by converting their operands to 16-bit, signed, two's complement integers in the range of −32768 to +32767. If the operands are not in the range an error message results. Each bit of the result is determined by the corresponding bits in the two operands.

EXAMPLES of OR Operator:

```
100 IF (AA = BB) OR (XX = 20) THEN . . . .

230 KK% = 64 OR 32: PRINT KK%        (You typed this with a bit
                                      value of 1000000 for 64
                                      and 100000 for 32)

96                                   (The computer responded
                                      with bit value 1100000.
                                      1100000=96.)
```

PEEK

TYPE: Integer Function
FORMAT: PEEK (<numeric>)

Action: Returns an integer in the range of 0 to 255, which is read from a memory location. The <numeric> expression is a memory location which must be in the range of 0 to 65535. If it isn't then the BASIC error message **?ILLEGAL QUANTITY** occurs.

EXAMPLES of PEEK Function:

```
10 PRINT PEEK(53280) AND 15          (Returns value of screen
                                      border color)
5 A%=PEEK(45)+PEEK(46)*256           (Returns address of BASIC
                                      variable table)
```

POKE

TYPE: Statement
FORMAT: POKE <location>, <value>

Action: The POKE statement is used to write a one-byte (8 bits) binary value into a given memory location or input/output register. The <location> is an arithmetic expression which must equal a value in the range of 0 to 65535. The <value> is an expression which can be reduced to an integer value of 0 to 255. If either value is out of its respective range, the BASIC error message **?ILLEGAL QUANTITY** occurs.

The POKE statement and PEEK statement (which is a built-in function that looks at a memory location) are useful for data storage, controlling graphics displays or sound generation, loading assembly language subroutines, and passing arguments and results to and from assembly language subroutines. In addition, Operating System parameters can be examined using PEEK statements or changed and manipulated using POKE statements. A complete memory map of useful locations is given in Appendix G.

EXAMPLES of POKE Statement:

```
POKE 1024, 1        (Puts an "A" at position 1 on the screen)
POKE 2040, PTR      (Updates Sprite #0 data pointer)
10 POKE RED, 32
20 POKE 36879, 8
2050 POKE A, B
```

POS

TYPE: Integer Function
FORMAT: POS (<dummy>)

Action: Tells you the current cursor position which, of course, is in the range of 0 (leftmost character) though position 79 on an 80-character logical screen line. Since the Commodore 64 has a 40-column screen, any position from 40 through 79 will refer to the second screen line. The dummy argument is ignored.

EXAMPLE of POS Function:

```
1000 IF POS(0) > 38 THEN PRINT CHR$(13)
```

PRINT

TYPE: Statement
FORMAT: PRINT [<variable>] [<,/;><variable>] . . . .

Action: The PRINT statement is normally used to write data items to the screen. However, the CMD statement may be used to re-direct that output to any other device in the system. The <variable(s)> in the output-list are expressions of any type. If no output-list is present, a blank line is printed. The position of each printed item is determined by the punctuation used to separate items in the output-list.

The punctuation characters that you can use are blanks, commas, or semicolons. The 80-character logical screen line is divided into 8 print zones of 10 spaces each. In the list of expressions, a comma causes the next value to be printed at the beginning of the next zone. A semicolon causes the next value to be printed immediately following the previous value. However, there are two exceptions to this rule:

1) Numeric items are followed by an added space.
2) Positive numbers have a space preceding them.

When you use blanks or no punctuation between string constants or variable names it has the same effect as a semicolon. However, blanks between a string and a numeric item or between two numeric items will stop output without printing the second item.

If a comma or a semicolon is at the end of the output-list, the next PRINT statement begins printing on the same line, and spaced accordingly. If no punctuation finishes the list, a carriage-return and a line-feed are printed at the end of the data. The next PRINT statement will begin on the next line. If your output is directed to the screen and the data printed is longer than 40 columns, the output is continued on the next screen line.

There is no statement in BASIC with more variety than the PRINT statement. There are so many symbols, functions, and parameters associated with this statement that it might almost be considered as a language of its own within BASIC; a language specially designed for writing on the screen.

EXAMPLES of PRINT Statement:

1)

```
 5 X = 5
10 PRINT -5*X, X-5, X+5, X ↑ 5

-25          0           10          3125
```

2)

```
 5 X=9
10 PRINT X;"SQUARED IS";X*X;"AND";
20 PRINT X "CUBED IS" X ↑ 3

 9 SQUARED IS 81 AND 9 CUBED IS 729
```

3)

```
 90 AA$="ALPHA":BB$="BAKER": CC$="CHARLIE":DD$="DOG":
    EE$="ECHO"
100 PRINT AA$BB$;CC$ DD$,EE$

ALPHABAKERCHARLIEDOG        ECHO
```

Quote Mode

Once the quote mark (SHIFT 2) is typed, the cursor controls stop operating and start displaying reversed characters which actually stand for the cursor control you are hitting. This allows you to program these cursor controls, because once the text inside the quotes is PRINTed they perform their functions. The INST/DEL key is the only cursor control not affected by "quote mode."

1. Cursor Movement

The cursor controls which can be "programmed" in quote mode are:

KEY	APPEARS AS
CLR/HOME	S
SHIFT CLR/HOME	♥
↑ CRSR ↓	Q
SHIFT ↑ CRSR ↓	○
← CRSR →	]
SHIFT ← CRSR →	▌

If you wanted the word HELLO to PRINT diagonally from the upper left corner of the screen, you would type:

PRINT " CLR/HOME H ↑ CRSR ↓ E ↑ CRSR ↓ L ↑ CRSR ↓ L ↑ CRSR ↓ O"

which would appear as:

PRINT " S H Q E Q L Q L Q O"

2. Reverse Characters

Holding down the CTRL key and hitting 9 will cause R to appear inside the quotes. This will make all characters start printing in ***reverse video*** (like a negative of a picture). To end the reverse printing hit CTRL Ø , which prints a ▭ or else PRINT a RETURN (CHR$(13)). (Just ending the PRINT statement without a semicolon or comma will take care of this.)

3. Color Controls

Holding down the CTRL key or C= key with any of the 8 color keys will make a special reversed character appear in the quotes. When the character is PRINTed, then the color change will occur.

KEY	COLOR	APPEARS AS
CTRL 1	Black	
CTRL 2	White	
CTRL 3	Red	
CTRL 4	Cyan	
CTRL 5	Purple	
CTRL 6	Green	
CTRL 7	Blue	
CTRL 8	Yellow	
C= 1	Orange	
C= 2	Brown	
C= 3	Light Red	
C= 4	Grey 1	
C= 5	Grey 2	
C= 6	Light Green	
C= 7	Light Blue	
C= 8	Grey 3	

If you wanted to PRINT the word HELLO in cyan and the word THERE in white, type:

PRINT " CTRL 4 HELLO CTRL 2 THERE"

which would appear as:

PRINT "  HELLO E THERE"

4. Insert Mode

The spaces created by using the INST/DEL key have some of the same characteristics as quote mode. The cursor controls and color controls show up as reversed characters. The only difference is in the INST and DEL , which performs its normal function even in quote mode, now

creates the T . And INST , which created a special character in quote mode, inserts spaces normally.

Because of this, it is possible to create a PRINT statement containing DELetes, which cannot be PRINTed in quote mode. Here is an example of how this is done:

10 PRINT"HELLO" INST/DEL SHIFT INST/DEL SHIFT INST/DEL INST/DEL INST/DEL P"

which displays as

10 PRINT"HELLO T T P"

When the above line is RUN, the word displayed will be HELP, because the last two letters are deleted and the P is put in their place.

WARNING: The DELetes will work when LISTing as well as PRINTing, so editing a line with these characters will be difficult.

The "insert mode" condition is ended when the RETURN (or SHIFT RETURN) key is hit, or when as many characters have been typed as spaces were inserted.

5. Other Special Characters

There are some other characters that can be PRINTed for special functions, although they are not easily available from the keyboard. In order to get these into quotes, you must leave empty spaces for them in the line, hit RETURN or SHIFT RETURN , and go back to the spaces with the cursor controls. Now you must hit CTRL RVS/ON , to start typing reversed characters, and type the keys shown below:

Function	Type	Appears As
SHIFT RETURN	SHIFT M	▚
switch to lower case	N	N
switch to upper case	SHIFT N	▞
disable case-switching keys	H	H
enable case-switching keys	I	I

The SHIFT RETURN will work in the LISTing as well as PRINTing, so editing will be almost impossible if this character is used. The LISTing will also look very strange.

PRINT#

TYPE: I/O Statement
FORMAT: PRINT#<file-number> [<variable>]
[<,/;><variable>] . . . .

Actions: The PRINT# statement is used to write data items to a logical file. It must use the same number used to OPEN the file. Output goes to the device-number used in the OPEN statement. The <variable> expressions in the output-list can be of any type. The punctuation characters between items are the same as with the PRINT statement and they can be used in the same ways. The effects of punctuation are different in two significant respects.

When PRINT# is used with tape files, the comma, instead of spacing by print zones, has the same effect as a semicolon. Therefore, whether blanks, commas, semicolons or no punctuation characters are used between data items, the effect on spacing is the same. The data items are written as a continuous stream of characters. Numeric items are followed by a space and, if positive, are preceded by a space.

If no punctuation finishes the list, a carriage-return and a line-feed are written at the end of the data. If a comma or semicolon terminates the output-list, the carriage-return and line-feed are suppressed. Regardless of the punctuation, the next PRINT# statement begins output in the next available character position. The line-feed will act as a stop when using the INPUT# statement, leaving an empty variable when the next INPUT# is executed. The line-feed can be suppressed or compensated for as shown in the examples below.

The easiest way to write more than one variable to a file on tape or disk is to set a string variable to CHR$(13), and use that string in between all the other variables when writing the file.

EXAMPLES of PRINT# Statement:

1)

```
10 OPEN 1,1,1, "TAPE FILE"
20 R$ = CHR$(13)
30 PRINT# 1,1;R$;2;R$;3;R$;4;R$;5
40 PRINT# 1,6
50 PRINT# 1,7
```

(By Changing the CHR$(13) to CHR$(44) you put a "," between each variable. CHR$(59) would put a ";" between each variable.)

2)

```
10 CO$=CHR$(44): CR$=CHR$(13)
20 PRINT#1, "AAA"CO$"BBB",
   "CCC";"DDD";"EEE"CR$
   "FFF"CR$;
30 INPUT#1, A$,BCDE$,F$
```

AAA,BBB CCCDDDEEE
(carriage return)
FFF(carriage return)

3)

```
 5 CR$=CHR$(13)
10 PRINT#2, "AAA";CR$;"BBB"
20 PRINT#2, "CCC";

30 INPUT#2, A$,B$,DUMMY$,C$
```

(10 blanks) AAA
BBB
(10 blanks)CCC

READ

TYPE: Statement
FORMAT: READ <variable> [,<variable>] . . . .

Action: The READ statement is used to fill variable names from constants in DATA statements. The data actually read must agree with the variable types specified or the BASIC error message **?SYNTAX ERROR** will result. Variables in the DATA input-list must be separated by commas.

A single READ statement can access one or more DATA statements, which will be accessed in order (see DATA), or several READ statements can access the same DATA statement. If more READ statements are executed than the number of elements in DATA statements(s) in the pro-

gram, the BASIC error message **?OUT OF DATA** is printed. If the number of variables specified is fewer than the number of elements in the DATA statement(s), subsequent READ statements will continue reading at the next data element. *(See RESTORE.)*

> *NOTE: The **?SYNTAX ERROR** will appear with the line number from the DATA statement, NOT the READ statement.

EXAMPLES of READ Statement:

```
110 READ A,B,C$
120 DATA 1,2,HELLO

100 FOR X=1 TO 10: READ A(X):NEXT

200 DATA 3.08, 5.19, 3.12, 3.98, 4.24
210 DATA 5.08, 5.55, 4.00, 3.16, 3.37
```

(Fills array items (line 1) in order of constants shown (line 5))

```
1 READ CITY$,STATE$,ZIP

5 DATA DENVER,COLORADO, 80211
```

REM

TYPE: Statement
FORMAT: REM [<remark>]

Action: The REM statement makes your programs more easily understood when LISTed. It's a reminder to yourself to tell you what you had in mind when you were writing each section of the program. For instance, you might want to remember what a variable is used for, or some other useful information. The REMark can be any text, word, or character including the colon (:) or BASIC keywords.

The REM statement and anything following it on the same line-number are ignored by BASIC, but REMarks are printed exactly as entered when the program is listed. A REM statement can be referred to by a GOTO or GOSUB statement, and the execution of the program will continue with the next higher program line having executable statements.

EXAMPLES of REM Statement:

```
10 REM CALCULATE AVERAGE VELOCITY
20 FOR X=1 TO 20 :REM LOOP FOR TWENTY VALUES
30 SUM=SUM + VEL(X): NEXT
40 AVG=SUM/20
```

RESTORE

TYPE: Statement
FORMAT: RESTORE

Action: BASIC maintains an internal pointer to the next DATA constant to be READ. This pointer can be reset to the first DATA constant in a program using the RESTORE statement. The RESTORE statement can be used anywhere in the program to begin re-READing DATA.

EXAMPLES of RESTORE Statement:

```
100 FOR X=1 TO 10: READ A(X): NEXT
200 RESTORE
300 FOR Y=1 TO 10: READ B(Y): NEXT

4000 DATA 3.08, 5.19, 3.12, 3.98, 4.24
4100 DATA 5.08, 5.55, 4.00, 3.16, 3.37
```

(Fills the two arrays with identical data)

```
10 DATA 1,2,3,4
20 DATA 5,6,7,8
30 FOR L=1 TO 8
40 READ A: PRINT A
50 NEXT
60 RESTORE
70 FOR L=1 TO 8
80 READ A: PRINT A
90 NEXT
```

RETURN

TYPE: Statement
FORMAT: RETURN

Action: The RETURN statement is used to exit from a subroutine called for by a GOSUB statement. RETURN restarts the rest of your program at the next executable statement following the GOSUB. If you are nesting subroutines, each GOSUB must be paired with at least one RETURN statement. A subroutine can contain any number of RETURN statements, but the first one encountered will exit the subroutine.

EXAMPLE of RETURN Statement:

```
10 PRINT "THIS IS THE PROGRAM"
20 GOSUB 1000
30 PRINT "PROGRAM CONTINUES"
40 GOSUB 1000
50 PRINT "MORE PROGRAM"
60 END
1000 PRINT "THIS IS THE GOSUB":RETURN
```

RIGHT$

TYPE: String Function
FORMAT: RIGHT$ (<string>, <numeric>)

Action: The RIGHT$ function returns a sub-string taken from the right-most end of the <string> argument. The length of the sub-string is defined by the <numeric> argument which can be any integer in the range of 0 to 255. If the value of the numeric expression is zero, then a null string ("") is returned. If the value you give in the <numeric> argument is greater than the length of the <string> then the entire string is returned.

EXAMPLE of RIGHT$ Function:

```
10 MSG$ = "COMMODORE COMPUTERS"
20 PRINT RIGHT$(MSG$,9)
RUN

 COMPUTERS
```

RND

TYPE: Floating-Point Function
FORMAT: RND (<numeric>)

Action: RND creates a floating-point random from 0.0 to 1.0. The computer generates a sequence of random numbers by performing calculations on a starting number, which in computer jargon is called a seed. The RND function is seeded on system power-up. The <numeric> argument is a dummy, except for its sign (positive, zero, or negative).

If the <numeric> argument is positive, the same "pseudorandom" sequence of numbers is returned, starting from a given seed value. Different number sequences will result from different seeds, but any sequence is repeatable by starting from the same seed number. Having a known sequence of "random" numbers is useful in testing programs.

If you choose a <numeric> argument of zero, then RND generates a number directly from a free-running hardware clock (the system "jiffy clock"). Negative arguments cause the RND function to be re-seeded with each function call.

EXAMPLES of RND Function:

```
220 PRINT INT(RND(0)*50)                    (Return random integers
                                             0-49)

100 X=INT(RND(1)*6)+INT(RND(1)*6)+2         (Simulates 2 dice)

100 X=INT(RND(1)*1000)+1                    (Random integers from
                                             1-1000)

100 X=INT(RND(1)*150)+100                   (Random numbers from
                                             100-249)

100 X=RND(1)*(U-L)+L                        (Random numbers between
                                             upper (U) and lower
                                             (L) limits)
```

RUN

TYPE: Command
FORMAT: RUN [<line-number>]

Action: The system command RUN is used to start the program currently in memory. The RUN command causes an implied CLR operation to be performed before starting the program. You can avoid the CleaRing operation by using CONT or GOTO to restart a program instead of RUN. If a <line-number> is specified, your program will start on that line. Otherwise, the RUN command starts at first line of the program. The RUN command can also be used within a program. If the <line-number> you specify doesn't exist, the BASIC error message **UNDEF'D STATEMENT** occurs.

A RUNning program stops and BASIC returns to direct mode when an END or STOP statement is reached, when the last line of the program is finished, or when a BASIC error occurs during execution.

EXAMPLES of RUN Command:

```
RUN       (Starts at first line of program)
RUN 500   (Starts at line-number 500)
RUN X     (Starts at line X, or UNDEF'D STATEMENT ERROR
           if there is no line X)
```

SAVE

TYPE: Command
FORMAT: SAVE ["<file-name>"] [,<device-number>] [,<address>]

Action: The SAVE command is used to store the program that is currently in memory onto a tape or diskette file. The program being SAVEd is only affected by the command while the SAVE is happening. The program remains in the current computer memory even after the SAVE operation is completed until you put something else there by using another command. The file type will be "prg" (program). If the <device-number> is left out, then the C64 will automatically assume that you want the program saved on cassette, device number 1. If the <device-number> is an <8>, then the program is written onto disk. The SAVE

statement can be used in your programs and execution will continue with the next statement after the SAVE is completed.

Programs on tape are automatically stored twice, so that your Commodore 64 can check for errors when LOADing the program back in. When saving programs to tape, the <file-name> and secondary <address> are optional. But following a SAVE with a program name in quotes (" ") or by a string variable (---$) helps your Commodore 64 find each program more easily. If the file-name is left out it can NOT be LOADed by name later on.

A secondary address of 1 will tell the KERNAL to LOAD the tape at a later time, with the program currently in memory instead of the normal 2048 location. A secondary address of 2 will cause an end-of-tape marker to follow the program. A secondary address of 3 combines both functions.

When saving programs onto a disk, the <file-name> must be present.

EXAMPLES of SAVE Command:

SAVE	(Write to tape without a name)
SAVE "ALPHA", 1	(Store on tape as file-name "alpha")
SAVE "ALPHA", 1, 2	(Store "alpha" with end-of-tape marker)
SAVE "FUN.DISK",8	(SAVES on disk (device 8 is the disk))
SAVE A$	(Store on tape with the name A$)
10 SAVE "HI"	(SAVEs program and then move to next program line)
SAVE "ME",1,3	(Stores at same memory location and puts an end-of-tape marker on)

SGN

TYPE: Integer Function
FORMAT: SGN (<numeric>)

Action: SGN gives you an integer value depending upon the sign of the <numeric> argument. If the argument is positive the result is 1, if zero the result is also 0, if negative the result is −1.

EXAMPLE of SGN Function:

```
90 ON SGN(DV)+2 GOTO 100, 200, 300
(jump to 100 if DV=negative, 200 if DV=0, 300 if DV=positive)
```

SIN

TYPE: Floating-Point Function
FORMAT: SIN (<numeric>)

Action: SIN gives you the sine of the <numeric> argument, in radians. The value of COS(x) is equal to SIN(x+3.14159265/2).

EXAMPLE of SIN Function:

```
235 AA = SIN(1.5): PRINT AA
 .997494987
```

SPC

TYPE: Special Function
FORMAT: SPC (<numeric>)

Action: The SPC function is used to control the formatting of data, as either an output to the screen or into a logical file. The number of SPaCes given by the <numeric> argument are printed, starting at the first available position. For screen or tape files the value of the argument is in the range of 0 to 255 and for disk files up to 254. For printer files, an automatic carriage-return and line-feed will be performed by the printer if a SPaCe is printed in the last character position of a line. No SPaCes are printed on the following line.

EXAMPLE of SPC Function:

```
10 PRINT "RIGHT "; "HERE &";
20 PRINT SPC(5) "OVER" SPC(14) "THERE"
RUN

RIGHT HERE &    OVER                THERE
```

SQR

TYPE: Floating-Point Function
FORMAT: SQR (<numeric>)

Action: SQR gives you the value of the SQuare Root of the <numeric> argument. The value of the argument must not be negative, or the BASIC error message **?ILLEGAL QUANTITY** will happen.

EXAMPLE of SQR Function:

```
FOR J = 2 TO 5: PRINT J*5, SQR(J * 5): NEXT

 10    3.16227766
 15    3.87298335
 20    4.47213595
 25    5
       READY
```

STATUS

TYPE: Integer Function
FORMAT: STATUS

Action: Returns a completion STATUS for the last input/output operation which was performed on an open file. The STATUS can be read from any peripheral device. The STATUS (or simply ST) keyword is a

system defined variable-name into which the KERNAL puts the STATUS of I/O operations. A table of STATUS code values for tape, printer, disk and RS-232 file operations is shown below:

ST Bit Position	ST Numeric Value	Cassette Read	Serial Bus R/W	Tape Verify + Load
0	1		time out write	
1	2		time out read	
2	4	short block		short block
3	8	long block		long block
4	16	unrecoverable read error		any mismatch
5	32	checksum error		checksum error
6	64	end of file	EOI	
7	−128	end of tape	device not present	end of tape

EXAMPLES of STATUS Function:

```
10 OPEN 1, 4: OPEN 2, 8, 4, "MASTER FILE,SEQ,W"
20 GOSUB 100: REM CHECK STATUS
30 INPUT#2, A$, B, C
40 IF STATUS AND 64 THEN 80: REM HANDLE END-OF-FILE
50 GOSUB 100: REM CHECK STATUS
60 PRINT#1, A$, B; C
70 GOTO 20
80 CLOSE1: CLOSE2
90 GOSUB 100: END
100 IF ST > 0 THEN 9000: REM HANDLE FILE I/O ERROR
110 RETURN
```

STEP

TYPE: Statement
FORMAT: [STEP <expression>]

Action: The optional STEP keyword follows the <end-value> expression in a FOR statement. It defines an increment value for the loop counter variable. Any value can be used as the STEP increment. Of course, a STEP value of zero will loop forever. If the STEP keyword is left out, the increment value will be +1. When the NEXT statement in a FOR loop is reached, the STEP increment happens. Then the counter is tested against the end-value to see if the loop is finished. (See FOR statement for more information.)

> **NOTE:** The STEP value can NOT be changed once it's in the loop.

EXAMPLES of STEP Statement:

```
25 FOR XX = 2 TO 20 STEP 2          (Loop repeats 10 times)

35 FOR ZZ = 0 TO -20 STEP -2        (Loop repeats 11 times)
```

STOP

TYPE: Statement
FORMAT: STOP

Action: The STOP statement is used to halt execution of the current program and return to *direct mode*. Typing the RUN/STOP key on the keyboard has the same effect as a STOP statement. The BASIC error message **?BREAK IN LINE nnnnn** is displayed on the screen, followed by **READY.** The "nnnnn" is the line-number where the STOP occurs. Any open files remain open and all variables are preserved and can be examined. The program can be restarted by using CONT or GOTO statements.

EXAMPLES of STOP Statement:

```
10 INPUT#1, AA, BB, CC
20 IF AA = BB AND BB = CC THEN STOP
30 STOP

(If the variable AA is -1 and BB is equal to CC then:)
BREAK IN LINE 20
BREAK IN LINE 30      (For any other data values)
```

STR$

TYPE: String Function
FORMAT: STR$ (<numeric>)

Action: STR$ gives you the STRing representation of the numeric value of the argument. When the STR$ value is converted to each variable represented in the <numeric> argument, any number shown is followed by a space and, if it's positive, it is also preceded by a space.

EXAMPLE of STR$ Function:

```
100 FLT = 1.5E4: ALPHA$ = STR$(FLT)
110 PRINT FLT, ALPHA$

 15000     15000
```

SYS

TYPE: Statement
FORMAT: SYS <memory-location>

Action: This is the most common way to mix a BASIC program with a machine language program. The machine language program begins at the location given in the SYS statement. The system command SYS is used in either direct or program mode to transfer control of the microprocessor to an existing machine language program in memory. The memory-location given is by numeric expression and can be anywhere in memory, RAM or ROM.

When you're using the SYS statement you must end that section of machine language code with an RTS (ReTurn from Subroutine) instruction so that when the machine language program is finished, the BASIC execution will resume with the statement following the SYS command.

EXAMPLES of SYS Statement:

```
SYS 64738                    (Jump to System Cold Start in ROM)

10 POKE 4400,96: SYS 4400    (Goes to machine code location 4400
                              and returns immediately)
```

TAB

TYPE: Special Function
FORMAT: TAB (<numeric>)

Action: The TAB function moves the cursor to a relative SPC move position on the screen given by the <numeric> argument, starting with the left-most position of the current line. The value of the argument can range from 0 to 255. The TAB function should only be used with the PRINT statement, since it has no effect if used with PRINT# to a logical file.

EXAMPLE of TAB Function:

```
100 PRINT "NAME" TAB(25) "AMOUNT": PRINT
110 INPUT#1, NAM$, AMT$
120 PRINT NAM$ TAB(25) AMT$

NAME                    AMOUNT

G.T. JONES              25.
```

TAN

TYPE: Floating-Point Function
FORMAT: TAN (<numeric>)

Action: Returns the tangent of the value of the <numeric> expression in radians. If the TAN function overflows, the BASIC error message **?DIVISION BY ZERO** is displayed.

EXAMPLE of TAN Function:

```
10 XX = .785398163: YY = TAN(XX): PRINT YY
 1
```

TIME

TYPE: Numeric Function
FORMAT: TI

Action: The TI function reads the interval TImer. This type of "clock" is called a "jiffy clock." The "jiffy clock" value is set at zero (initialized) when you power-up the system. This 1/60 second interval timer is turned off during tape I/O.

EXAMPLE of TI Function:

```
10 PRINT TI/60 "SECONDS SINCE POWER UP"
```

TIME$

TYPE: String Function
FORMAT: TI$

Action: The TI$ timer looks and works like a *real clock* as long as your system is powered-on. The hardware interval timer (or jiffy clock) is read and used to update the value of TI$, which will give you a TIme $tring of six characters in hours, minutes and seconds. The TI$ timer can also be assigned an arbitrary starting point similar to the way you set your wristwatch. The value of TI$ is not accurate after tape I/O.

EXAMPLE of TI$ Function:

```
1 TI$ = "000000": FOR J=1 TO 10000: NEXT: PRINT TI$

000011
```

USR

TYPE: Floating-Point Function
FORMAT: USR (<numeric>)

Action: The USR function jumps to a User callable machine language SubRoutine which has its starting address pointed to by the contents of memory locations 785–786. The starting address is established before calling the USR function by using POKE statements to set up locations 785–786. Unless POKE statements are used, locations 785–786 will give you an **?ILLEGAL QUANTITY** error message.

The value of the <numeric> argument is stored in the floating-point accumulator starting at location 97, for access by the Assembler code, and the result of the USR function is the value which ends up there when the subroutine returns to BASIC.

EXAMPLES of USR Function:

```
10 B = T * SIN(Y)
20 C = USR (B/2)
30 D = USR (B/3)
```

VAL

TYPE: Numeric Function
FORMAT: VAL (<string>)

Action: Returns a numeric VALue representing the data in the <string> argument. If the first non-blank character of the string is not a plus sign (+), minus sign (−), or a digit the VALue returned is zero. String conversion is finished when the end of the string or any non-digit character is found (except decimal point or exponential e).

EXAMPLE of VAL Function:

```
10 INPUT#1, NAM$, ZIP$
20 IF VAL(ZIP$) < 19400 OR VAL(ZIP$) > 96699
   THEN PRINT NAM$ TAB(25) "GREATER PHILADELPHIA"
```

VERIFY

TYPE: Command
FORMAT: VERIFY ["<file-name>"] [,<device>]

Action: The VERIFY command is used, in direct or program mode, to compare the contents of a BASIC program file on tape or disk with the program currently in memory. VERIFY is normally used right after a SAVE, to make sure that the program was stored correctly on tape or disk.

If the <device> number is left out, the program is assumed to be on the Datassette™ which is device number 1. For tape files, if the <file-name> is left out, the next program found on the tape will be compared. For disk files (device number 8), the file-name must be present. If any differences in program text are found, the BASIC error message **?VERIFY ERROR** is displayed.

A program name can be given either in quotes (" ") or as a string variable. VERIFY is also used to position a tape just past the last program, so that a new program can be added to the tape without accidentally writing over another program.

EXAMPLES of VERIFY Command:

```
VERIFY                          (Checks 1st program on tape)
PRESS PLAY ON TAPE
OK
SEARCHING
FOUND <FILENAME>
VERIFYING

9000 SAVE "ME",8:
9010 VERIFY "ME",8              (Looks at device 8 for the program)
```

WAIT

TYPE: Statement
FORMAT: WAIT <location>, <mask-1> [,<mask-2>]

Action: The WAIT statement causes program execution to be suspended until a given memory address recognizes a specified bit pattern. In other words WAIT can be used to halt the program until some external event has occurred. This is done by monitoring the status of bits in the input/output registers. The data items used with WAIT can be any numeric expressions, but they will be converted to integer values.

For most programmers, this statement should never be used. It causes the program to halt until a specific memory location's bits change in a specific way. This is used for certain I/O operations and almost nothing else.

The WAIT statement takes the value in the memory location and performs a logical AND operation with the value in mask-1. If there is a mask-2 in the statement, the result of the first operation is exclusive-ORed with mask-2. In other words mask-1 "filters out" any bits that you don't want to test. Where the bit is 0 in mask-1, the corresponding bit in the result will always be 0. The mask-2 value flips any bits, so that you can test for an off condition as well as an on condition. Any bits being tested for a 0 should have a 1 in the corresponding position in mask-2.

If corresponding bits of the <mask-1> and <mask-2> operands differ, the exclusive-OR operation gives a bit result of 1. If corresponding bits get the same result the bit is 0. It is possible to enter an infinite pause with the WAIT statement, in which case the RUN/STOP and RESTORE keys can be used to recover. Hold down the RUN/STOP key and then press RESTORE. The first example below WAITs until a key is pressed on the tape unit to continue with the program. The second example will WAIT until a sprite collides with the screen background.

EXAMPLES of WAIT Statement:

```
WAIT 1, 32, 32
WAIT 53273, 6, 6
WAIT 36868, 144, 16     (144 & 16 are masks. 144=10010000 in
                         binary and 16=10000 in binary. The
                         WAIT statement will halt the pro-
                         gram until the 128 bit is on or
                         until the 16 bit is off)
```

THE COMMODORE 64 KEYBOARD AND FEATURES

The Operating System has a ten-character keyboard "buffer" that is used to hold incoming keystrokes until they can be processed. This buffer, or queue, holds keystrokes in the order in which they occur so that the first one put into the queue is the first one processed. For example, if a second keystroke occurs before the first can be processed, the second character is stored in the buffer, while processing of the first character continues. After the program has finished with the first character, the keyboard buffer is examined for more data, and the second keystroke processed. Without this buffer, rapid keyboard input would occasionally drop characters.

In other words, the keyboard buffer allows you to "type-ahead" of the system, which means it can anticipate responses to INPUT prompts or GET statements. As you type on the keys their character values are lined up, single-file (queued) into the buffer to wait for processing in the order the keys were struck. This type-ahead feature can give you an occasional problem where an accidental keystroke causes a program to fetch an incorrect character from the buffer.

Normally, incorrect keystrokes present no problem, since they can be corrected by the CuRSoR-Left ←CRSR or DELete INST/DEL keys and then retyping the character, and the corrections will be processed before a following carriage-return. However, if you press the RETURN key, no corrective action is possible, since all characters in the buffer up to and including the carriage-return will be processed before any corrections. This situation can be avoided by using a loop to empty the keyboard buffer before reading an intended response:

```
10 GET JUNK$: IF JUNK$ <>"" THEN 10: REM EMPTY THE KEYBOARD BUFFER
```

In addition to GET and INPUT, the keyboard can also be read using PEEK to fetch from memory location 197 ($00C5) the integer value of the key currently being pressed. If no key is being held when the PEEK is executed, a value of 64 is returned. The numeric keyboard values, keyboard symbols and character equivalents (CHR$) are shown in Appendix C. The following example loops until a key is pressed then converts the integer to a character value.

```
10 AA = PEEK(197): IF AA = 64 THEN 10
20 BB$ = CHR$(AA)
```

The keyboard is treated as a set of switches organized into a matrix of 8 columns by 8 rows. The keyboard matrix is scanned for key switch-closures by the KERNAL using the CIA #1 I/O chip (MOS 6526 Complex Interface Adapter). Two CIA registers are used to perform the scan: register #0 at location 56320 ($DC00) for keyboard columns and register #1 at location 56321 ($DC01) for keyboard rows.

Bits 0–7 of memory location 56320 correspond to the columns 0–7. Bits 0–7 of memory location 56321 correspond to rows 0–7. By writing column values in sequence, then reading row values, the KERNAL decodes the switch closures into the CHR$ (N) value of the key pressed.

Eight columns by eight rows yields 64 possible values. However, if you first strike the RVS, CTRL or C= keys or hold down the SHIFT key and type a second character, additional values are generated. This is because the KERNAL decodes these keys separately and "remembers" when one of the control keys was pressed. The result of the keyboard scan is then placed in location 197.

Characters can also be written directly to the keyboard buffer at locations 631–640 using a POKE statement. These characters will be processed when the POKE is used to set a character count into location 198. These facts can be used to cause a series of direct-mode commands to be executed automatically by printing the statements onto the screen, putting carriage returns into the buffer, and then setting the character count. In the example below, the program will LIST itself to the printer and then resume execution.

```
10 PRINT CHR$(147)"PRINT#1: CLOSE 1: GOTO 50"
20 POKE 631,19: POKE 632,13: POKE 633,13: POKE 198,3
30 OPEN 1,4: CMD1: LIST
40 END
50 REM PROGRAM RE-STARTS HERE
```

SCREEN EDITOR

The SCREEN EDITOR provides you with powerful and convenient facilities for editing program text. Once a section of a program is listed to the screen, the cursor keys and other special keys are used to move around the screen so that you can make any appropriate changes. After making all the changes you want to a specific line-number of text, hitting the RETURN key anywhere on the line, causes the SCREEN EDITOR to read the entire 80-character logical screen line.

The text is then passed to the Interpreter to be tokenized and stored in the program. The edited line replaces the old version of that line in memory. An additional copy of any line of text can be created simply by changing the line-number and pressing RETURN.

If you use keyword abbreviations which cause a program line to exceed 80 characters, the excess characters will be lost when that line is edited, because the EDITOR will read only two physical screen lines. This is also why using INPUT for more than a total of 80 characters is not possible. Thus, for all practical purposes, the length of a line of BASIC text is limited to 80 characters as displayed on the screen.

Under certain conditions the SCREEN EDITOR treats the *cursor control* keys differently from their normal mode of handling. If the CuRSoR is positioned to the *right* of an *odd number* of double-quote marks (") the EDITOR operates in what is known as the QUOTE-MODE.

In *quote mode* data characters are entered normally but the cursor controls no longer move the CuRSoR, instead reversed characters are displayed which actually stand for the cursor control being entered. The same is true of the color control keys. This allows you to include cursor and color controls inside string data items in programs. You will find that this is a very important and powerful feature. That's because when the text inside the quotes is printed to the screen it performs the cursor positioning and color control functions automatically as part of the string. An example of using cursor controls in strings is:

You type → 10 PRINT "A(R)(R)B(L)(L)(L)C(R)(R)D":REM(R)=CRSR RIGHT, (L)=CRSR LEFT

Computer prints → AC BD

The DEL key is the only cursor control NOT affected by quote mode. Therefore, if an error is made while keying in quote mode, the ←CRSR key can't be used to back up and strike over the error—even the INST key produces a reverse video character. Instead, finish entering the line, and then, after hitting the RETURN key, you can edit the line normally. Another alternative, if no further cursor-controls are needed in the string, is to press the RUN/STOP and RESTORE keys which will cancel QUOTE MODE. The cursor control keys that you can use in strings are shown in Table 2-2.

Table 2–2. Cursor Control Characters in QUOTE MODE

Control Key		Appearance
CRSR up	SHIFT CRSR	○
CRSR down	CRSR	Q
CRSR left	SHIFT CRSR	▌
CRSR right	CRSR	]
CLR	CLR/	♥
HOME	/HOME	S
INST	INST/DEL	▐

When you are NOT in quote mode, holding down the SHIFT key and then pressing the INSerT INST key shifts data to the right of the cursor to open up space between two characters for entering data between them. The Editor then begins operating in *INSERT MODE* until all of the space opened up is filled.

The cursor controls and color controls again show as reversed characters in insert mode. The only difference occurs on the DELete and INSerT INST/DEL key. The DEL instead of operating normally as in the quote mode, now creates the reversed T . The INST key, which created a reverse character in quote mode, inserts spaces normally.

Ths means that a PRINT statement can be created, containing DELetes, which can't be done in quote mode. The insert mode is cancelled by pressing the RETURN , SHIFT and RETURN , or RUN/STOP and RESTORE keys. Or you can cancel the insert mode by filling all the inserted spaces. An example of using DEL characters in strings is:

```
10 PRINT "HELLO" DEL INST INST DEL DEL P"
(Keystroke sequence shown above, appearance when listed below)
10 PRINT"HELP"
```

When the example is RUN, the word displayed will be HELP, because the letters LO are deleted before the P is printed. The DELete character in strings will work with LIST as well as PRINT. You can use this to "hide" part or all of a line of text using this technique. However, trying to edit a line with these characters will be difficult if not impossible.

There are some other characters that can be printed for special functions, although they are not easily available from the keyboard. In order to get these into quotes, you must leave empty spaces for them in the line, press RETURN, and go back to edit the line. Now you hold down the CTRL (ConTRoL) key and type RVS/ON (ReVerSe-ON) to start typing reversed characters. Type the keys as shown below:

Key Function	Key Entered	Appearance
Shifted RETURN	SHIFT M	
Switch to upper/lower case	N	N
Switch to upper/graphics	SHIFT N	

Holding down the SHIFT key and hitting RETURN causes a carriage-return and line-feed on the screen but does not end the string. This works with LIST as well as PRINT, so editing will be almost impossible if this character is used. When output is switched to the printer via the CMD statement, the reverse "N" character shifts the printer into its upper-lower case character set and the SHIFT "N" shifts the printer into the upper-case/graphics character set.

Reverse video characters can be included in strings by holding down the ConTRoL CTRL key and pressing ReVerSe RVS, causing a reversed R to appear inside the quotes. This will make all characters print in reverse video (like a negative of a photograph). To end the reverse printing, press CTRL and RVS/OFF (ReVerSe OFF) by holding down the CTRL key and typing the RVS/OFF key, which prints a reverse R. Numeric data can be printed in reverse video by first printing a CHR$(18). Printing a CHR$(146) or a carriage-return will cancel reverse video output.