ERROR

Format **ERROR <integer expression>**

Purpose Simulates the occurrence of a BASIC error. Also allows error codes to be defined by the user.

Remarks The value of <integer expression> must be greater than 0 and less than 255. If the value specified equals one of the error codes which is used by BASIC (see Appendix A), occurrence of that error is simulated and the corresponding error message is displayed. If the value specified is not defined in BASIC, the message "UL error" (Unprintable error) is displayed.

You can also use the ERROR statement to define your own error messages; this is illustrated in the example below. When using the ERROR statement for this purpose the value of <integer expression> must be a number which does not correspond to any error code which is defined in BASIC. Such user-defined error codes can then be handled in an error processing routine.

See also **ERR, ERL, ON ERROR GOTO, RESUME**

Example

```
10 ON ERROR GOTO 80
20 X = "FRED"
30 X$(20) = 23
40 ERROR 199
50 PRINT
60 PRINT "There are no more errors to demonstrate"
70 END
80 IF ERR = 13 THEN PRINT "There is a Type Mismatch in line
20":RESUME 30
90 IF ERL = 30 THEN PRINT "There is a Subscript error in lin
e 30":RESUME 40
100 IF ERR = 199 THEN PRINT "I defined this error number 199
 myself":RESUME 50

There is a Type Mismatch in line 20
There is a Subscript error in line 30
I defined this error number 199 myself

There are no more errors to demonstrate
Ok
```

ERR

Format **ERR**

Purpose Used in an error processing routine to return the code of an error occurring during program execution.

Remarks The ERR function returns the code of errors occurring during command or statement execution.

As with the ERL function, the ERR function is normally used with IF ... THEN statements in an error processing routine to control the flow of program execution when an error occurs. An example of program control using the ERR function is shown below.

See also **ERL, ON ERROR GOTO, RESUME**

Example See under ERROR

IF ERR = 11 THEN RESUME 1000

When included in an error processing routine, this line causes program execution to resume at line 1000 if the error being processed is a /0 error (Division by zero).

EXP

Format **EXP(X)**

Purpose Returns the value of the natural base *e* to the power of X.

Remarks The value specified for X must not be greater than 87.3365; otherwise an OV error (Overflow) will occur.

To raise another number to a power use the operator "^". See the program below for an example.
EXP can also be used to obtain antilogarithms.

See also **LOG**

Example

```
10 FOR J = 0 TO 80 STEP 10
20 LPRINT "e^";J;"=";EXP(J)
30 NEXT
40 LPRINT
50 LPRINT "The cube of 2 is: ";2^3
```

```
e^ 0 = 1
e^ 10 = 22026.5
e^ 20 = 4.85165E+08
e^ 30 = 1.06865E+13
e^ 40 = 2.35385E+17
e^ 50 = 5.1847E+21
e^ 60 = 1.142E+26
e^ 70 = 2.51544E+30
e^ 80 = 5.54063E+34

The cube of 2 is:  8
```

FIELD

Format **FIELD[#]<file number>, <field width> AS <string variable>, <field width> AS <string variable>, ...**

Purpose Assigns string variables to specific positions in a random file buffer.

Remarks When a random access file is opened, a buffer is automatically reserved in memory which is used for temporary storage of data while it is being transferred between the storage medium (RAM disk or other disk device) and the computer's memory. Data is read into this buffer from the storage medium during input, and is written from the buffer to the storage medium during output. During input, data is read into the buffer upon execution of a GET statement, and is output to the buffer upon execution of a PUT statement.

However, before any GET or PUT statement can be executed, a FIELD statement must be executed to assign positions in the file access buffer to specific variables. Doing this causes data substituted into that variable by a PUT statement to be stored in assigned positions in the file access buffer, rather than in normal string space. Conversely, items brought into the buffer from the file by a GET statement are accessed by checking the contents of variables to which positions in the buffer have been assigned. See Chapter 5 for a detailed description of procedures for accessing random access files.

The <file number> which is specified in the FIELD statement is that under which the file was opened. <field width> specifies the number of positions which is to be allocated to the specified <string variable>. For example:

FIELD 1,20 AS N$,10 AS ID$,40 AS ADD$

assigns the first 20 positions of the buffer to string variable N$, the next 10 positions to ID$, and the next 40 positions to ADD$.

The total number of positions assigned to variables by the FIELD statement cannot exceed the record length that was specified when the file was opened; otherwise, an FO error (Field overflow) will occur (the default record length is 128 bytes).

If necessary, any number of FIELD statements may be executed for the same file. If more than one such statement is executed, all assignments made are effective concurrently.

See also **GET, LSET/RSET, OPEN, PUT**
For full details of use of the FIELD command see Chapter 5.

Example

```
10 OPEN "R",#1,"E:EMPDAT.DAT"
20 '   Opens file "EMPDAT.DAT" in drive E: as a random
30 '   access file.
40 FIELD #1,6 AS NO$,2 AS DP$,120 AS RE$
50 '   Assigns the first six bytes of file buffer #1 to
60 '   variable NO$, the second two bytes to variable
70 '   DP$, and the last 120 bytes to variable RE$.
80 '   Data can now be placed in the buffer with the
90 '   LSET/RSET statement, then stored in the file with
100 '  the PUT statement. Or, data can be brought into
110 '  the buffer from the file with the GET statement,
120 '  after which the contents of the buffer variables
130 '  can be displayed with the PRINT statement or
140 '  used for other processing.
```

NOTE:
Once a variable name has been specified in a FIELD statement, only use RSET or LSET to store data in that variable. FIELDing a variable name assigns it to specific positions in the random file buffer; using an INPUT or LET statement to store values to FIELDed variables will cancel this assignment and reassign the names to normal string space.

FILES

Format **FILES[<ambiguous file name>]**

Purpose Displays the names of files satisfying the <ambiguous file name>.

Remarks If <ambiguous file name> is omitted, this command displays the names of all files in the disk device which is the currently selected drive.

When specified, <ambiguous file name> is composed of the following elements.

[<drive name>:][<file name>[<.extension>]]

In this case, the FILES command lists the names of all files which satisfy the <ambiguous file name>. An ambiguous file name is used to find files whose file names and/or extensions include common character strings. The form for specifying ambiguous file names is similar to that used for normal file descriptors, except that the question mark (?) can be used as a wild card character to indicate any character in a particular position, and the asterisk (*) can be used as a wild card character to indicate any combination of characters for a file name or extension. Some examples of ambiguous file names are shown below.

If there are no files on a particular disk, the "File not found" message will be displayed.

Example 1 **FILES "A:L???.BAS"**
Displays all files on disk drive A: whose file names begin with L, are up to four characters long and are also BASIC files.

Example 2 **FILES "L???????.BAS"**
Displays the names of all files on the currently active disk device whose file names begin with the letter L and whose extensions are ".BAS", because all possible character positions are used with the '?' wild card.

Example 3 **FILES "D:*.*" or FILES "D:"**
Displays the names of all files on the disk in drive D.

Example 4 **FILES "D:D???.*"**
Displays the names of all files on the disk in drive D which begin with the letter D and which include not more than four characters in the file name.

Example 5 **FILES "D:*.COM"**
Displays all the files on disk drive D: which have .COM as their extension.

PRINTING FILE NAMES FROM BASIC
In order to print the directory of a disk from BASIC, a screen dump must be carried out. If there are more files than the screen will show at one time, scroll the screen using the cursor keys and perform multiple screen dumps. Selecting screen size of 40 × 50 by WIDTH command is the best to use for screen dumps of files because it can be scrolled, and will hold the greatest number of file names at any one time.

FIX

Format **FIX(X)**

Purpose Returns the integer portion of numeric expression X.

Remarks The value returned by FIX(X) is equal to the sign of X times the integer portion of the absolute value of X. Thus −1 is returned for −1.5, −2 is returned for −2.33333 and so forth. Compare with the INT function, which returns the largest integer which is less than or equal to X.

See CINT for an explanation and comparison of CINT, FIX and INT, and also other problems associated with their use.

FONT

Format **FONT [STEP](X,Y),<function>,<user defined character code>[,<user defined character code>......]**

Purpose Displays user defined characters at the specified location.

Remarks This statement displays user defined characters assigned to the specified user defined character codes. The coordinates (X,Y) indicate a point which is the upper left corner of the first character displayed. Y is an integer from 0 to 13 but X must be a multiple of 8 (0, 8, 16,...) between 0 and 232. If X is other than multiples of 8, the greatest one of multiples of 8 which is less than X is used as the X coordinate.
When the specified location is out of the window, the specified characters are not displayed or part of the characters is displayed.

<function> is one of the following.

PSET:	Displays characters as they are.
PRESET:	Displays reversed characters.
OR:	Displays the result of logical sum (OR) of the dot pattern existing on the screen and that of the specified characters.
AND:	Displays the result of logical product (AND) of the dot pattern existing on the screen and that of the specified characters.
XOR:	Displays the result of XOR (exclusive OR) operation on the dot pattern existing on the screen and that of the specified characters.

FOR...NEXT

Format **FOR <variable> = <expression 1> TO <expression 2> [STEP <expression 3>]**

```
·
·
·
```

NEXT [<variable>][,<variable>...]

Purpose The FOR...NEXT statement allows the series of instructions between FOR and NEXT to be repeated a specific number of times.

Remarks This statement causes program execution to loop through the series of instructions between FOR and NEXT a specific number of times. The number of repetitions is determined by the values specified following FOR for <expression 1>, <expression 2>, and <expression 3>.

<variable> is used as a counter for keeping track of the number of loops which have been made. The initial value of this counter is that specified by <expression 1>. The ending value of the counter is that specified by <expression 2>. Program lines following FOR are executed until the NEXT statement is encountered, then the counter is incremented by the amount specified by <expression 3>. An increment of 1 is assumed if STEP <expression 3> is not specified; however, a negative value must be specified following STEP if the value of <expression 2> is less than that of <expression 1>.

Next, a check is made to see if the value of the counter is greater than the value specified by <expression 2>. If not, execution branches back to the statement following the FOR statement and the sequence is repeated. If the value is greater, execution continues with the statement following NEXT. Otherwise statements in the loop are skipped and execution resumes with the first statement following NEXT.

FOR...NEXT loops may be nested; that is, one FOR...NEXT loop may be included within the body of another one. When loops are nested, different variable names must be specified for <variable> at the beginning of each loop. Further, the NEXT statement for inner loops must appear before those for outer ones.

If nested loops end at the same point, a single NEXT statement may be used for all of them. In this case, the variable names must be specified following NEXT in reverse order to that in which they appear in the FOR statements of the nested loops; in other words, the first variable name following NEXT must be that which is specified in the nearest preceding FOR statement, the second variable name following NEXT must be that which is specified in the next nearest preceding FOR statement, and so forth.

If a NEXT statement is encountered before its corresponding FOR statement, an NF error (NEXT without FOR) message is displayed and execution is aborted. If a FOR statement without a corresponding NEXT statement is encountered, an FN error (FOR without NEXT) message is displayed and execution is aborted.

See also **WHILE...WEND**

Example

```
10 PRINT "1. Single loop incremented in steps of 2"
20 FOR J = 1 TO 9 STEP 2
30 PRINT J,
40 NEXT
50 FOR L = 1 TO 100:NEXT
60 PRINT:PRINT "2. Single loop with default increment of 1"
70 FOR K = 1 TO 5
80 PRINT K,
90 NEXT
100 FOR L = 1 TO 100:NEXT
110 PRINT:PRINT "3. Nested loop with two NEXT statements"
120 FOR J = 1 TO 5
130 FOR K = 1 TO 3
140 PRINT J;"/";K,
150 NEXT:NEXT
160 FOR L = 1 TO 100:NEXT
170 PRINT:PRINT "4. Nested loop with one NEXT statement
    specifying both variable
180 FOR J = 1 TO 5
190 FOR K = 1 TO 3
200 PRINT J;"/";K,
210 NEXT K,J
```

```
1. Single loop incremented in steps of 2
 1            3            5            7            9

2. Single loop with default increment of 1
 1            2            3            4            5

3. Nested loop with two NEXT statements
 1 / 1        1 / 2        1 / 3        2 / 1        2 / 2
 2 / 3        3 / 1        3 / 2        3 / 3        4 / 1
 4 / 2        4 / 3        5 / 1        5 / 2        5 / 3

4. Nested loop with one NEXT statement specifying both
   variables
 1 / 1        1 / 2        1 / 3        2 / 1        2 / 2
 2 / 3        3 / 1        3 / 2        3 / 3        4 / 1
 4 / 2        4 / 3        5 / 1        5 / 2        5 / 3
```

FRE

Format

FRE(X)
FRE(X$)

Purpose

Returns the number of bytes of memory which are not being used by BASIC.

Remarks

The value returned by this function provides an indication of the number of bytes of memory which are available for use as string variables, numeric variables or BASIC program text. However, the value returned also includes a work area which is used by BASIC during program execution and thus does not provide a direct indication of the number of unused bytes which are available for this purpose.

The arguments of this function have no meaning; FRE returns the same value regardless of whether a string expression or a numeric expression is specified as the argument.

When a FRE(X$) function is executed, the BASIC interpreter has to perform 'garbage collection'. When strings are defined BASIC stores them in memory; they are often changed frequently as the program is executed. Every so often BASIC has to collect the values of the strings which are still valid and erase the unwanted ones, otherwise there will be no space for more strings to be stored. If a program executes a great deal of string handling there can be times while the program is running that it appears to halt because it is carrying out this 'garbage collection'. By executing FRE(X$), BASIC is forced to carry out this operation. A line which checks the free memory is often placed throughout a BASIC program so that by forcing garbage collection in small steps rather than one big one the long gap can be avoided.

FRE(X) carries out garbage collection for numerical variables in a manner similar to that in which FRE(X$) performs garbage collection for string variables.

Example

The following program shows how assigning a value to a variable decreases the amount of free memory available.

Note that when the program is run, line 10 shows there are 23665 bytes of memory. When line 20 has allocated a value to variable B, available memory is reduced to 23657 bytes. After string manipulation, line 70 shows the memory is reduced a great deal further. In executing the FRE(X$) command, much of this memory can be retrieved.

```
10 PRINT "Free memory for programs and variables using FRE(X)
is";FRE(X)
20 B= 10
30 PRINT "Free memory for programs and variables using FRE(X)
is";FRE(X)
40 FOR J = 65 TO 75
50 A$ = A$ + CHR$(J)
60 NEXT J
70 PRINT "Free memory for programs and variables using FRE(X)
is";FRE(X)
80 PRINT "and using FRE(X$) is ";FRE(X$)

run
Free memory for programs and variables using FRE(X) is 23665
Free memory for programs and variables using FRE(X) is 23657
Free memory for programs and variables using FRE(X) is 23565
and using FRE(X$) is  23631
Ok
```

GET

Format **GET[#]<file number>[,<record number>]**

Purpose The GET statement reads a record into a random file access buffer from a random disk file.

Remarks This statement reads a record into a random file access buffer from the corresponding random access file. <file number> is the number under which the file was opened and <record number> is the number of the record which is to be read into the random file buffer. Both <file number> and <record number> must be specified as integer expressions.

If <record number> is omitted the record read is that following the one read by the preceding GET statement. The highest possible record number is 32767.

Note that records must be read sequentially if the file being accessed is a microcassette file. Also note that the FIELD statement must be executed to assign space in the random file buffer to variables prior to executing a GET statement.

See also **FIELD, LSET/RSET, OPEN, PUT**
For full details of use of the FIELD command see Chapter 4.

Example

```
10 'Lines 20-80 create a random data file with 5 records.
20 OPEN"R",#1,"A:TESTDAT",10
30 FIELD#1,10 AS A$
40 FOR I=1 TO 5
50 PRINT"Type 1-10 characters for record";I;:INPUT B$
60 LSET A$=B$
70 PUT#1,I
80 NEXT
90 '
100 'Lines 110 to read specified records from the file.
110 INPUT"Enter record no.(1-5)";R
120 GET#1,R
130 PRINT A$
140 GOTO 110
```

```
run
Type 1-10 characters for record 1 ? Alfie
Type 1-10 characters for record 2 ? Betty
Type 1-10 characters for record 3 ? Charlie
Type 1-10 characters for record 4 ? Dean
Type 1-10 characters for record 5 ? Edward
Enter record no.(1-5)?


Enter record no.(1-5)? 5
Edward
Enter record no.(1-5)? 4
Dean
Enter record no.(1-5)? 3
Charlie
Enter record no.(1-5)? 2
Betty


Enter record no.(1-5)? 1
Alfie
Enter record no.(1-5)?
```

GOSUB...RETURN

Format

GOSUB <line number>

•

•

RETURN

Purpose

The GOSUB and RETURN statements are used to branch to and return from subroutines.

Remarks

The GOSUB statement transfers execution to the program line number specified in <line number>. When a RETURN statement is encountered, execution then returns to the statement following the one which called the subroutine, either on the same line or the next one. Subroutines may be located anywhere in a program; however, it is recommended that they be made readily distinguishable from the main routine. Since an RG error (RETURN without GOSUB) will occur if a RETURN statement is encountered without a corresponding GOSUB statement, care must be taken to ensure that execution does not move into a subroutine without it having been called. This can be avoided with the STOP, END or GOTO statements; the STOP and END statements halt execution when encountered, while the GOTO statement can be used to route execution around the subroutine.

Subroutines may include more than one RETURN statement if the program logic dictates a return from different points in the subroutine. Further, a subroutine may be called any number of times in a program, and one subroutine may be called by another. Nesting of subroutines in this manner is limited only by the amount of stack space available for storing return addresses. An OM error (Out of memory) will occur if the stack space is exceeded. The stack space size may be changed with the CLEAR statement if it is insufficient to accomodate the number of levels of subroutine nesting used by a program, but this must be executed at the beginning of the program outside the subroutines as CLEAR destroys all references to RETURN line numbers on the stack and you will not be able to RETURN from a subroutine if CLEAR is used within it.

See also **CLEAR**

Example

```
10 GOSUB 70
20 PRINT "Resuming execution after returning from subroutine
at line 70"
30 GOSUB 60
40 PRINT "Resuming execution after return from the nested
subroutines startin at line 50"
50 END
60 GOSUB 70:RETURN
70 PRINT "Now executing the subroutine at line 70"
80 RETURN

run
Now executing the subroutine at line 70
Resuming execution after returning from subroutine at line 70
Now executing the subroutine at line 70
Resuming execution after return from the nested subroutines st
arting at line 50
Ok
```

GOTO or GO TO

Format

GOTO <line number>
GO TO <line number>

Purpose

Unconditionally transfers program execution to the program line specified by <line number>.

Remarks

This statement is used to make unconditional "jumps" from one point in a program to another. If the first statement on the line specified by <line number> is an executable statement (other than a REM or DATA statement), execution resumes with that statement; otherwise, execution resumes with the first executable statement encountered following <line number>.

It is also possible to leave out the GOTO in conditional statements, e.g., line 20 in the following program.

A UL error (Undefined line number) will occur if <line number> refers to a non-existent line.

GOTO can also be used in direct mode. In this case, variables are not destroyed (unlike RUN <line number> which does destroy variables), and can in fact be assigned from the command line in the direct mode.

Example

```
10 READ A,B:'Reads numbers into A and B from line 70
20 IF A=0 AND B=0 THEN 80:'Jumps to line 80 if A and B
25 '                          both equal 0.
30 PRINT "A=";A,"B=";B
40 S=A*B
50 PRINT "Product is";S
60 GOTO 10                  :'Jumps to line 10.
70 DATA 12,5,8,3,9,0,0,0
80 END

run
A= 12          B= 5
Product is 60
A= 8           B= 3
Product is 24
A= 9           B= 0
Product is 0
Ok
```

HEX$

Format

HEX$(X)

Purpose

Returns a character string which represents the hexadecimal value of X.

Remarks

The value of the numeric expression specified in the argument must be a number in the range from −32768 to 65535. If the value of the expression includes a decimal fraction, it is rounded to the nearest integer before the string representing the hexadecimal value is returned.

To convert from hexadecimal to decimal use &H before the hexadecimal value. This will give a numerical constant.

HEX$ is a string.
&H is a numeric.

See also

OCT$

Example 1

```
10 CLS
20 LOCATE 1,1: PRINT "Convert Hex to Decimal (H) or ":
PRINT "Decimal to Hex (D)"
30 INPUT C$
40 IF C$ = "H" OR C$ = "h" THEN GOSUB 100 ELSE IF C$ =
"D" OR C$ = "d" THEN GOSUB 200 ELSE 20
50 INPUT "Any more (yes/no) "; YN$
60 IF LEFT$(YN$,1) <> "y" AND LEFT$(YN$,1) <> "Y" THEN
END ELSE RUN
100 INPUT "Type in number in hexadecimal ";H$
110 V  = VAL ("&H" + H$)
120 PRINT V
130 RETURN
200 INPUT "Type in number in decimal "; D
210 V$ = HEX$ (D)
220 PRINT V$
230 RETURN
```

```
run
Convert Hex to Decimal (H) or
Decimal to Hex (D)
? h
Type in number in hexadecimal ? 4d
 77
Any more (yes/no) ?


Convert Hex to Decimal (H) or
Decimal to Hex (D)
? d
Type in number in decimal ? 34
22
Any more (yes/no) ?
```

IF...THEN [...ELSE]/IF...GOTO

Format Possible alternatives are

IF <logical expression> THEN <statement> [ELSE <statement>]

IF <logical expression> THEN <line No.> [ELSE <line No.>]

IF <logical expression> THEN <statement> [ELSE <line No.>]

IF <logical expression> THEN <line No.> [ELSE <statement>]

IF <logical expression> GOTO <line No.> [ELSE <statement>]

IF <logical expression> GOTO <line No.> [ELSE <line No.>]

Purpose Changes the flow of program execution according to the results of a logical expression.

Remarks The THEN or GOTO clause following <logical expression> is executed if the result of <logical expression> is true (– 1). Otherwise, the THEN or GOTO clause is ignored and the ELSE clause (if any) is executed; execution then proceeds with the next executable statement.

When a THEN clause is specified, THEN may be followed by either a line number or one or more statements. Specifying a line number following THEN causes program execution to branch to that program line in the same manner as with GOTO. When a GOTO clause is specified, GOTO is always followed by a line number.

IF...THEN...ELSE statements may be nested by including one such statement as a clause in another. Such nesting is limited only by the maximum length of the program line.

For example, the following is a correctly nested IF...THEN statement

20 IF X>Y THEN PRINT "X IS LARGER THAN Y" ELSE IF Y>X THEN PRINT "X IS SMALLER THAN Y" ELSE PRINT "X EQUALS Y"

Because of the logical structure of the line only one of the strings can be printed.

If a statement contains more THEN than ELSE clauses, each ELSE clause is matched with the nearest preceding THEN clause. For example, the following statement displays "A=C" when A=B and B=C. If A=B and B< >C it will display "A< >C". And if A< >B, it displays nothing at all.

IF A=B THEN IF B=C THEN PRINT "A=C" ELSE PRINT "A< >C"

It is also possible to have a number of statements where <statement> occurs in the above format expressions. For example it is common to have a line such as:

IF A=2 THEN B=3:C=7:A$="ORANGE"

Only if A has the value 2 will B be set equal to 3. The value of C will also only be set equal to 7 and A$ given the value "ORANGE" if A=2. If the expression "A=2" is false then all the rest of the line will be ignored.

This also applies if the sequence of statements exists in a line such as the following:

IF A=2 THEN PRINT "TRUE":B=5:A$="APPLES": GOTO 200 ELSE PRINT "FALSE":B=7:A$="PEARS": GOTO 300

In this case if A has the value 2 the expression "A=2" is true and the variable B is made equal to 5, the value "APPLES" is assigned to A$ and the program branches to line 200. However, if A does not have the value 2, the expression "A=2" is false and the commands following ELSE are executed; B is set equal to 7, A$ is assigned the value "PEARS" and the program will branch to line 300.

When using IF together with a relational expression which tests for equality, remember that the results of arithmetic operations are not always exact values. For example, the result of the relational expression SIN(1.5708)=1 is false even though "1" is displayed if PRINT SIN(1.5708) is executed. Therefore, the relational expression should be written in such a way that computed values are tested over the range within which the accuracy of such values may vary. For example, if you are testing for equality between SIN(1.5708) and 1, the following form is recommended:

IF ABS(1 – SIN(1.5708))<1.0E – 6 THEN...

This relational expression returns "true".

Example

```
10 SCREEN 0,0,0:CLS
20 RANDOMIZE VAL(RIGHT$(TIME$,2))
30 '     Reinitializes the sequence of numbers returned
40 '     by the RND function
50 '
60 PRINT "Guess what number I am thinking of."
70 '
80 N=INT(RND(1)*9999)
90 '      Generates a random 4-digit number between
100 '     0 and 9999 and stores it in variable N
110 '
120 INPUT"Enter your guess";G
130 I=I+1
140 '     Keeps track of the number of guesses
150 '
160 IF G=N THEN PRINT "That's just right--in";I;"guesses!":E
LSE IF G<N THEN PRINT "You're too low! Try again.":GOTO 120:
ELSE PRINT "Sorry--you're too high! Try again.":GOTO 120
170 '     Displays the first message and the number
180 '     of guesses you have made if you correctly
190 '     guess the number generated on line 40; if
200 '     your guess is too low, displays the second
210 '     message and branches to line 50; if your
220 '     guess is too high, displays the third
230 '     message and branches to line 50
```

INKEY$

Format INKEY$

Purpose Checks the keyboard buffer during program execution and returns a null string if no key has been pressed.

Remarks INKEY$ returns a null string if the keyboard buffer is empty. If any key whose code is included in the ASCII code table has been pressed, INKEY$ reads that character from the keyboard buffer and returns it to the program. Characters read from the keyboard buffer by INKEY$ are not displayed on the screen.

INKEY$ simply examines the keyboard buffer. It does not wait for a key to be pressed. If this function is required, use INPUT$(1).

See also INPUT$

Example

```
10 SCREEN 0,0,0
20 WIDTH 80,8:CLS
30 X=1:Y=1:LOCATE X,Y:PRINT "*";:'Displays an asterisk
40 '                               in the upper left
50 '                               corner of the screen.
60 '
70 A$=INKEY$:IF A$="" THEN 70
80 '    Checks for input from the keyboard; repeats
90 '    until input is detected.
100 '
110 ON INSTR(CHR$(30)+CHR$(31)+CHR$(29)+CHR$(28),A$) GOSUB 2
50,300,350,400
120 '    Checks whether the key pressed is one of the
130 '    cursor control keys; if so, goes to the
140 '    corresponding subroutine.  Otherwise,
150 '    continues to the GOTO statement on the line
160 '    below.
170 '
180 GOTO 70:'Transfers execution to line 70.
190 '
200 'The four subroutines below move the asterisk
210 'in the direction indicated by the arrow on
220 'the applicable cursor key.
230 '
240 'Move asterisk up
```

```
250 Y=Y-1:IF Y<1 THEN Y=8
260 LOCATE X,Y:PRINT "*";:IF Y=8 THEN LOCATE X,1:PRINT " ":E
LSE LOCATE X,Y+1:PRINT " "
270 RETURN
280 '
290 'Move asterisk down
300 Y=Y+1:IF Y>8 THEN Y=1
310 LOCATE X,Y:PRINT "*";:IF Y=1 THEN LOCATE X,8:PRINT " ";:
ELSE LOCATE X,Y-1:PRINT " "
320 RETURN
330 '
340 'Move asterisk left
350 X=X-1:IF X<1 THEN X=80
360 LOCATE X,Y:PRINT "*";:IF X=80 THEN LOCATE 1,Y:PRINT " ";
:ELSE LOCATE X+1,Y:PRINT " ";
370 RETURN
380 '
390 'Move asterisk right
400 X=X+1:IF X>80 THEN X=1
410 LOCATE X,Y:PRINT "*";:IF X=1 THEN LOCATE 80,Y:PRINT " ";
:ELSE LOCATE X-1,Y:PRINT " ";
420 RETURN
```

INP

Format
INP (J)

Purpose
Returns one byte of data from machine port J.

Remarks
The machine port number specified for J must be an integer expression in the range from 0 to 255.

The full use of this command is beyond the scope of this manual. Please see the System Documentation for details of the ports.

See also
OUT

Example

```
10 'See I/O port &H02 if LED is ON
20 A=INP(&H2)
30 LED=A AND &H4      :'See BIT 2 status
40 IF LED=0 GOTO 60   :'If LED is OFF, set LED
50 END
60 OUT &H2,&H4        :'Set LED
```

INPUT

Format
INPUT[;"<prompt string>"] |;| <list of variables>
|,|

Purpose
Makes it possible to substitute values into variables from the keyboard during program execution.

Remarks
Program execution pauses when an INPUT statement is encountered to allow data to be substituted into variables from the keyboard. One data item must be typed in for each variable name specified in <list of variables>, and each item typed in must be separated from the following one by a comma. If any commas are to be included in a string substituted into a given variable, that string must be enclosed in quotation marks when it is typed in from the keyboard. The same applies to leading and trailing spaces; leading and trailing spaces are not substituted into a string variable by the INPUT statement unless the string is enclosed in quotation marks.

If the number or type of items entered is incorrect, the message "?Redo from start" is displayed, followed by the prompt string (if any). When this occurs, all the data items must be re-entered. No values are substituted into variables until a correct response has been made to all the items of the list.

When BASIC executes the line a question mark prompt is displayed if no prompt string is specified. However, if a prompt string is specified, the string is displayed, but whether a question mark is displayed depends on the character before the list of variables. It is possible to enter a comma or a semi-colon before the list of variables. In the latter case, a question mark will be displayed. If the prompt string is followed by a comma, the question mark is suppressed. If no prompt string is given it is not possible to suppress the question mark.

The optional semicolon following INPUT prevents the cursor from advancing to the next line when the user types a RETURN on completion of entry of the data. The next PRINT statement or error statement will be printed directly after the last character input by the user before pressing RETURN .

When more than one variable name is specified in <list of variables>, each variable name must be separated from the following one by a comma. Items entered in response to an INPUT statement are substituted into the variables specified in <list of variables> when the RETURN key is pressed. The user must input each variable and separate it from the following one by a comma. If the user tries to press RETURN after each variable, a "?Redo from start" message will be printed, and the user will have to begin at the first item of the list of variables. The values entered are only substituted into the variables when the RETURN key is pressed at the end of the list.

When the RETURN key is pressed for a single item or for the last item of a list, the variable is set to a null string if it is a string variable and to zero if it is a numeric variable.

Example

```
10 INPUT A                  :'Inputs value from keyboard
20                          :'into A, then moves cursor
30                          :'to next line.
40                          :'
50 INPUT;B                  :'Inputs value into B and
60                          :'keeps cursor on current
70                          :'line.
80                          :'
90 INPUT"Enter C",C         :'Displays prompt without
100                         :'question mark and inputs
110                         :'value into C.
120                         :'
130 INPUT"Enter D,E";D,E    :'Displays prompt and
140                         :'question mark and inputs
150                         :'values into D and E.
160                         :'
170 INPUT;"Enter F,G";F,G:'Displays prompt, inputs
180                         :'values into F and G, and
190                         :'keeps cursor on current line.
200 PRINT "END"
```

INPUT

Format **INPUT # <file number>, <variable list>**

Purpose This statement is used to read items from a sequential disk file in a similar way to that in which the INPUT statement reads data from the keyboard.

Remarks The sequential file from which data is to be read with this statement must have been previously opened for input by executing an OPEN statement. <file number> is the number under which the file was opened.

As with INPUT, <variable list> specifies the names of variables into which items of data are to be read when the INPUT # statement is executed. Variables specified must be of the same type as data items which are read. Otherwise, a "Type mismatch" error will occur.

Upon execution of this statement, data items are read in from the file in sequence until one item has been assigned to each variable in <variable list>. When the file is read with this statement, the first character encountered which is not a space is assumed to be the start of a data item. With string items, the end of one item is assumed when the following character is a comma or a carriage return, however, individual string items may include commas and carriage returns if they are enclosed in quotation marks when they were saved to the file. The end of a data item is also assumed if 255 characters are read without encountering a comma or carriage return.

With numeric items, the end of each item is assumed when a space, comma, or carriage return is encountered. Therefore, care must be taken to ensure that proper delimiters are used when the file is written to the disk file with the PRINT # statement.

Examples of use of the INPUT # statement are shown in the program below.

See also **INPUT, LINE INPUT, LINE INPUT #, OPEN, PRINT #, WRITE, WRITE #**
Chapter 4

Example

```
10 OPEN "O",#1,"a:test1.dat"
20 FOR I=1 TO 16:'    Saves numeric data items "1"
30 '                  to "16" to file "a:test1.dat"
40 PRINT#1,I
50 NEXT I
60 PRINT#1,"a";CHR$(13);"b"
70 '   Saves "a" and "b" to file "a:test1.dat"
80 '   as separate data items.  Items are separated
90 '   by a carriage return code (CHR$(13)).
100 PRINT#1,CHR$(34)+"c,d,e"+CHR$(34)
110 '    Saves "c,d,e" to file "a:test1.dat" as one
120 '    data item.  This is regarded as one item because
130 '    it is enclosed in quotation marks (CHR$(34))
140 '    to indicate that the commas are part of the
150 '    string, and not delimiting characters.
160 PRINT#1,"f,g"
170 '    Saves "f,g" to file "a:test1.dat" as separate
180 '    data items.  The reason for this is that commas
190 '    are regarded as delimiters unless quotation marks
200 '    are saved to the disk to indicate that the commas
210 '    are part of a string.  Quotation marks are saved
220 '    to a file by specifying their ASCII codes with
230 '    the CHR$ function as shown above with "c,d,e".
240 CLOSE
250 DIM A(15)
260 OPEN "I",#1,"a:test1.dat"
270 FOR I=0 TO 15
280 INPUT#1,A(I)
290 NEXT I
300 FOR I=0 TO 15
310 PRINT A(I);
320 NEXT
330 PRINT
340 INPUT#1,A$,B$,C$,D$,E$
350 PRINT A$:PRINT B$:PRINT C$:PRINT D$:PRINT E$
360 CLOSE

run
 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16
a
b
c,d,e
f
g
Ok
```

INPUT$

Format **INPUT$(X[,[#]<file number>])**

Purpose Reads a string of X characters from the keyboard buffer or the file opened under <file number>.

Remarks INPUT$(X) reads the number of characters specified by X from the keyboard buffer and returns a string consisting of those characters to the program. If the keyboard buffer does not contain the specified number of characters, INPUT$ reads those characters which are present and waits for other keys to be pressed. Characters read are not displayed on the screen.

Unlike the INPUT and LINE INPUT statements, INPUT$ can be used to pass control characters such as RETURN (character code 13) to the program.

INPUT$(X,[#]<file number>) reads the number of characters specified by X from a sequential file opened under the specified file number. As with the first format, characters which would be recognized as delimiters between items by the INPUT# or LINE INPUT# statements are returned as part of the character string.

Execution of the INPUT$ function can be terminated by pressing the STOP key.

The BASIC statement

A$=INPUT$(1)

is useful for waiting for a single key to be pressed, in contrast to

100 A$=INKEY$: IF INKEY$=" "THEN 100

whereas INKEY$ can scan the keyboard buffer simply to test if a key has been pressed without waiting for it to be pressed.

Example

```
10 OPEN"O",#1,"test"
20 PRINT#1,"ABCDEFGHIJKLMNOPQRSTUVWXYZ"
30 CLOSE
40 OPEN"I",#1,"test"
50 '
60 '
100 A$=INPUT$(10,#1)  :'Inputs 10 characters into
110                   :'A$ from sequential file opened
120                   :'under file number 1.
125 PRINT A$
130                   :'
140 B$=INPUT$(10)     :'Inputs 10 characters into B$
150                   :'from keyboard.
160 PRINT B$

run
ABCDEFGHIJ
qwertyuiop
Ok
```

INSTR

Format **INSTR([J,]X$,Y$)**

Purpose Searches for the first occurrence of string Y$ in string X$ and returns the position at which a match is found.

Remarks If J is specified, the search for string Y$ begins at position J in string X$. J must be specified as an integer expression in the range from 1 to 255; otherwise, an "Illegal function call" error will occur. If a null string is specified for Y$, INSTR returns the value which is equal to that specified for J. A value of "0" is returned if J is greater than the length of X$, if X$ is a null string, or if Y$ cannot be found. Both X$ and Y$ may be specified as string variables, string expressions or string literals.

Example

```
10 'Example of using INSTR with ON...GOTO to control
20 'flow of program execution.
30 '
40 INPUT"Enter a,b,or c";X$
50 ON INSTR(1,"abc",X$) GOTO 70,90,110
60 PRINT"Illegal entry, try again.":GOTO 40
70 PRINT"Character entered is ";CHR$(34);"a.";CHR$(34)
80 END
90 PRINT"Character entered is ";CHR$(34);"b.";CHR$(34)
100 END
110 PRINT"Character entered is ";CHR$(34);"c.";CHR$(34)
120 END

run
Enter a,b,or c? a
Character entered is "a."
Ok
```

INT

Format INT(X)

Purpose Returns the largest integer which is less than or equal to X.

Remarks Any numeric expression may be specified for X.

See also CINT, FIX

The explanation of CINT also contains information on the differences between CINT, FIX and INT and describes why some problems arise from conversion and storage of numbers in connection with these functions.

Example

```
10 PRINT "I","INT(I)"
20 FOR I=-5 TO 5 STEP .8
30 PRINT I,INT(I)
40 NEXT I
I             INT(I)
-5            -5
-4.2          -5
-3.4          -4
-2.6          -3
-1.8          -2
-1            -1
-.2           -1
 .6            0
 1.4           1
 2.2           2
 3             3
 3.8           3
 4.6           4
```

KEY

Format **KEY <programable function key no.>,<character string>**
KEY <item function key no.>,<character string>
KEY <item function key switch>
KEY LIST
KEY LLIST

Purpose Used to define the programmable function keys and item function keys. Also used to output a list of character strings assigned to all function keys to the screen or printer.

Remarks The first two formats indicated above assign the specified contents of <character string> to the function key specified in <programmable function key no.> or <item function key no.>. For the programmable function keys, the function key number is specified as a number from 1 to 10. For the item function keys, the function key number is specified as a number from &H40 to &H7E.

<character string> is specified as any combination of up to 15 characters. If more than 15 characters are specified in <character string>, the 16th and following characters are ignored when the statement is executed.

The third format indicated above is used to clear, disable, or enable the item function keys. All item functions are cancelled when 255 is specified in <item function key switch>, all item function keys are disabled when 254 is specified, and all item function keys are enabled when 253 is specified. However, the item function keys can be temporarily enabled after executing KEY 254 by pressing them together with the CTRL and SHIFT keys. The KEY LIST statement outputs the definitions of the programmable function keys to the display screen, and the KEY LLIST outputs a similar list to the printer.

Initial definitions of the programmable function keys are as follows.

PF1	auto	PF6	load"
PF2	list	PF7	save"
PF3	edit	PF8	system
PF4	stat	PF9	menu^M
PF5	run^M	PF10	login_

FC error (Illegal function call) — The number specified in one of the statement operands was outside of the prescribed range.

MO error (Missing operand) — A required operand was not specified in the statement.

KILL

Format **KILL <file descriptor>**

Purpose Used to delete files from a disk device.

Remarks The KILL command can be used to delete any type of disk file. The full file descriptor must be specified if the file to be deleted is in a drive other than that which is currently selected. Otherwise, only the file name and extension need to be specified.

Example **KILL "A:GRAPH.BAS"**

This will delete the file in drive A: "GRAPH" which has the extension ".BAS".

NOTE:
Operation of the KILL command is not assured if it is issued against a file which is currently OPEN.

LEFT$

Format **LEFT$(X$,J)**

Purpose Returns a string composed of the J characters making up the left end of string X$.

Remarks The value specified for J must be in the range from 0 to 255. If J is greater than the length of string X$, the entire string will be returned. If J is zero, a null string of zero length will be returned.

See also **MID$, RIGHT$**

Example 1 A$ = LEFT$("CARROT",3) will return "CAR" to be stored in A$.

Example 2

```
10 A$ = "EPSON"
20 FOR J = 1 TO 6
30 PRINT LEFT$(A$,J)
40 NEXT
```

```
E
EP
EPS
EPSO
EPSON
EPSON
```

LEN

Format **LEN(X$)**

Purpose Returns the number of characters in string X$.

Remarks The number returned by this function also indicates any blanks or non-printable characters included in the string (such as the return and cursor control codes).

Example

```
10 CLS
20 INPUT "Type in a word or phrase";A$
30 PRINT "The length of :- "
40 PRINT A$
50 PRINT "is"; LEN(A$); "characters"
60 GOTO 20
```

```
Type in a word or phrase? FRED
The length of :-
FRED
is 4 characters
Type in a word or phrase?
```

```
Type in a word or phrase? CHARLIE IS SUPER
The length of :-
CHARLIE IS SUPER
is 16 characters
Type in a word or phrase?
```

LET

Format

[LET] <variable> = <expression>

Purpose

Assigns the value of <expression> to <variable>.

Remarks

Note that the word LET is optional. Thus, in the example below, the variables A$ and B$ give the same result when printed, as do A and B.

Example

```
10 CLS
20 LET A$ = "THIS IS A STRING"
30 B$ = "THIS IS A STRING"
40 PRINT A$
50 PRINT B$
60 LET A = 3*4
70 B = 3 * 4
80 PRINT A,B

THIS IS A STRING
THIS IS A STRING
 12             12
Ok
```

LINE

Format

LINE[[STEP] (X1,Y1)] – [STEP](X2,Y2)[,[<function code>] [,[B[F]][,<line style>]]]

Purpose

Draws a line between two specified points.

Remarks

This statement is a graphics command which can only be used for graphic screen (LCD display). It draws a straight line between two specified points on the graphic screen. The coordinates of the first point are specified as (X1, Y1) and those of the second point are specified as (X2, Y2).

If STEP is omitted, (X1, Y1) and (X2, Y2) are absolute screen coordinates; if STEP is specified, (X1, Y1) indicates coordinates in relation to the last dot specified by the last graphic display statement executed (PSET, PRESET or LINE). The coordinates of the last previously specified dot are maintained by a pointer referred to as the last reference pointer (LRP); this pointer is updated automatically whenever a PSET, PRESET, or LINE statement is executed.

For example

LINE (0,0) – (239,63)
draws a line diagonally from the top left hand corner to the bottom right hand corner.

LINE – (100,50)
draws a line from the last plotted point (i.e. the LRP) to the point (100,50).

LINE (10,10) – STEP (100,30)
draws a line from point (10,10) to a point 100 points to the right and 30 down from point (10,10); i.e., to point (110,40).

LINE – STEP (100,50)
draws a line 100 points to the right and 50 points down from the coordinates of the LRP.

LINE STEP (10,10) – (100,50)
draws a line from a point 10 to the right and 10 down from the LRP to the absolute point (100,50).

LINE STEP (10,10) – STEP (100,40)
draws a line from a point 10 to the right and 10 down from the LRP. The LRP is then updated and the line drawn 100 points to the right and 50 down from the first end point of the line. Thus if the last point plotted before this command was executed was (5,3), the line would be drawn from the point (15,13) to (115,53).

<function code> is a number from 0 to 7 which specifies the line function. If 0 is specified, the LINE statement resets (turns off) dots along the line between the specified coordinates. If a number from 1 to 7 is specified, dots along the line are set (turned on) when the statement is executed. If no <function code> is specified, 7 is assumed.

Specifying "B" causes the LINE statement to draw a rectangle whose diagonal dimension is defined by the two points specified. If the F option is specified together with the B option, the rectangle is filled in. However, the BF option cannot be specified together with <line style>, although simply using B will allow rectangles to be drawn using different line types.

If you want to use the B or BF function without using the <function code>, a comma must be used as separator.

For example

LINE (0,0) — (20,15) ,,BF
will fill a box 20 points wide and 15 points high in the top left hand corner of the screen.

The <line style> option is a parameter which determines the type of line drawn between the two specified points. The line style is specified as any number which can be represented with 16 binary digits; i.e., the line style can be specified as any number from 0 to 65535 (in hexadecimal notation, from &H0 to &HFFFF). There is a one-to-one correspondence between the settings of the binary digits of <line style> and the settings of each 16-dot segment of the line drawn when the statement is executed. When the

<function code> is 1 to 7 or defaults to 7 because no value is inserted, all points corresponding to "1" bits are set (i.e., plotted). When the <function code> is specified as 0, all points corresponding to "1" bits are reset (i.e., erased). This is illustrated in the second example program below. In both cases where dots correspond to "0" bits no action is taken. When the length of the line is greater than 16 dots, the pattern is repeated for each 16-dot segment.

For example, dot settings are as follows when <line style> is specified as 1, 43690, and 61680.

<line style>	Binary equivalent	
1 (&H1)	0000000000000001	
	---------------*	Dot settings
		(* for on, – for off)
43690 (&HAAAA)	1010101010101010	
	--*-*-*-*-*-*-	Dot settings
61680 (&HF0F0)	1111000011110000	
	****----****----	Dot settings

The LINE statement can only be executed during display in the graphic mode (screen mode 3).

See also **PRESET, PSET**

LINE INPUT

Format **LINE INPUT[;][<prompt string>;]<string variable>**

Purpose Used to substitute string data including all punctuation into string variables from the keyboard during program execution.

Remarks The LINE INPUT statement is similar to the INPUT statement in that it is used to substitute values into variables from the keyboard during program execution.
However, whereas the INPUT statement can be used to input both numeric and string values, the LINE INPUT statement can only be used for input of string values. Further, only one such string can be input each time the LINE INPUT statement is executed. It is not possible to specify a list of variables separated by commas as is the case with the INPUT statement, because commas are accepted as part of the string.

INPUT allows commas to be entered if the first character typed is the quotation marks character. Quotation marks can be entered as long as they are not input as the first character. LINE INPUT on the other hand allows all characters to be substituted into the specified variable exactly as entered. Further, no question mark is displayed when a LINE INPUT statement is executed unless one has been included in <prompt string> by the user.

As with the INPUT statement, a semicolon immediately following LINE INPUT suppresses the carriage return typed by the user. The cursor is positioned after the last character entered by the user before pressing RETURN .

See also INPUT

Example

```
5 CLS
10 LINE INPUT "TYPE IN SOME CHARACTERS ";A$
15 PRINT " This is on the next line"
20 PRINT "The characters were ";A$
30 LINE INPUT;"AND SOME MORE ";B$
40 PRINT " This is on the same line"
50 PRINT "The characters were ";B$

TYPE IN SOME CHARACTERS CHARLIE IS MY DARLING
 This is on the next line
The characters were CHARLIE IS MY DARLING
AND SOME MORE OVER THE SEA TO SKYE This is on the same line
The characters were OVER THE SEA TO SKYE
Ok
```

LINE INPUT

Format

LINE INPUT # <file number>,<string variable>

Purpose

Used to read data into string variables from a sequential access file, in the same way that LINE INPUT is used to read strings from the keyboard.

Remarks

The LINE INPUT # statement is similar to the INPUT # statement in that it is used to read data into variables from a sequential access file. The value of <file number> is the number under which the file was opened, and <string variable> is the name of the variable into which data is read when the statement is executed.

Whereas the INPUT # statement can be used to read both numeric and string values, the LINE INPUT # statement can only be used to read character strings. Further, only one such string can be read each time the LINE INPUT # statement is executed. It is not possible to specify a list of variables, as is possible with the INPUT # statement.

Another difference between the INPUT # and LINE INPUT # statements is that whereas the former recognizes both commas and carriage returns as delimiters between data items, the LINE INPUT # statement regards all characters up to a carriage return (up to a maximum of 255 characters) as one data item. Any commas encountered are regarded as part of the string being read. The carriage return code itself is skipped, so the next LINE INPUT # statement begins reading data at the character following the carriage return.

This statement can be used to read all values written by a PRINT # statement into one variable. It also allows lines of a BASIC program which has been saved in ASCII format to be input as data by another program.

See also

INPUT #

Example

See Chapter 5.

LIST

Format

1) LIST[*] [[<line no. 1>][-[<line no. 2>]]
2) LIST[*] <file descriptor>
[,[<line no. 1>][-[<line no. 2>]]]
3) LLIST[*] [[<line no. 1>][-[<line no. 2>]]

Purpose

Outputs all or part of a program in memory to an external device.

Remarks

The LIST command outputs all or part of the program in the currently selected program area to the screen or other external device.With format 1) above, output is to the screen; with format 2), output is to the specified device; when it is omitted, output is to the display screen.

If <line no. 1> is specified by itself, only that line of the program is output.

If <line no. 1> and the hyphen are specified, the line specified and all following lines are output.

If the hyphen and <line no. 2> are specified, all lines from the beginning of the program to the specified line are output.

If both <line no. 1> and <line no. 2> are specified, all lines within that range are output.

When the asterisk is specified, program lines are output without line numbers.Further, any remark statements which are specified using an apostrophe (') are printed without the apostrophe, making it possible to use the LIST statement in simple word processing applications.

In the case of the LLIST command, output is always directed to the printer; however, in other respects it is exactly the same as the LIST command.

LLIST

Format **LLIST[*][<line number>][–<line number>]**

Purpose Lists all or part of the lines of the program in the currently logged in program area to a printer.

Remarks The LLIST command is used in the same manner as LIST, but output is always directed to the printer connected to the PX-4. BASIC always returns to the command level after execution of a LLIST command.

See also **LIST**

Example 1 **LLIST**
Prints all lines of the program in the currently logged in program area.

Example 2 **LLIST***
Same as above, but prints program lines without line numbers.

Example 3 **LLIST 500**
Prints program line 500.

Example 4 **LLIST 150-**
Prints all program lines from line 150 to the end of the program.

Example 5 **LLIST -1000**
Prints all lines from the beginning of the program to line 1000 (inclusive).

Example 6 **LLIST 150-1000**
Prints program lines from 150 to 1000 (inclusive).

LOAD

Format **LOAD<file descriptor>[,R]**

Purpose Loads a program into memory from a disk drive, RAM disk, the RS-232C interface, RAM cartridge, ROM cartridge, or the microcassette drive.

Remarks Specify the device name, file name, and extension under which the program was saved in <file descriptor>. If the device name is omitted, the currently selected drive is assumed; if the file name extension is omitted, ".BAS" is assumed.

When a LOAD command is executed without specifying the "R" option, all files which are open are closed, all variables are cleared, and all lines of any program in the currently logged in program area are cleared; after loading is completed, BASIC returns to the command level.

However, if the "R" is specified, any files which are currently open remain open and program execution begins as soon as loading has been completed. Thus, LOAD with the "R" option may be used to chain execution of programs which use the same data files. The following restrictions must be noted when using LOAD with the "R" option to chain execution of programs.

- All variables are cleared by execution of the LOAD command, regardless of whether the "R" option is specified. Further, the COMMON statement cannot be used to pass variables to the program called. Therefore, some other provision must be made for passing data to the program called (for example, intermediate data could be saved in a file in RAM disk).
- All assignments of variables to positions in random file buffers are cancelled even though the random access files to which the buffers belong remain open. Therefore, the FIELD statement must be executed in the called program to remake these assignments.

See also **CHAIN, MERGE, RUN, SAVE**

Example 1 **LOAD"A:PROG1.BAS"**

Example 2 (Example of program call using LOAD)

```
10 CLS
20 PRINT "This is the calling program, sometimes called the
loader"
30 PRINT "It will now load the program LOAD2.BAS...."
40 LOAD "A:LOAD2.BAS",R

This is the calling program, sometimes called the loader
It will now load the program LOAD2.BAS....
```

Example 3 (Example of program called by Example 2)

```
10 PRINT
20 PRINT"This is the program called LOAD2.BAS which has been
 loaded by the loading program LOAD1.BAS"
30 END

This is the program called LOAD2.BAS which has been loaded
by the loading program LOAD1.BAS
Ok
```

LOAD?

Format **LOAD? [<file descriptor>]**

Purpose Verifies the contents of a file.

Remarks This statement is used to check whether the file specified in <file descriptor> has been properly recorded. No device name other than CAS0: may be specified in <file descriptor>.

The LOAD? statement reads the contents of the specified file in the mode (stop or non-stop) in which it was written to the external audio cassette and verifies its contents by making CRC checks. If any CRC error is detected, an IO error (Device I/O error) occurs. Programs are not actually loaded by this statement, so the contents of memory remain unaffected.

When this statement is executed, all files preceding that specified in <file descriptor> are skipped; after the file is checked, the head of the external cassette recorder is positioned to the end of the file checked (to the beginning of the next file).

If <file descriptor> is omitted, the first file found is checked.