The standard form of the COMMON statement is referred to as blank COMMON. Microsoft FORTRAN Compiler-style named COMMON areas are also supported; however, the variables are not preserved across CHAINs. The syntax for named COMMON is:

```
COMMON /<name>/ <list of variables>
```

where <name> is comprised of 1 to 6 alphanumeric characters starting with a letter. This is useful for communicating with Microsoft FORTRAN Compiler and assembly language routines without having to explicitly pass parameters in the CALL statement.

The blank COMMON size and order of variables must be the same in the CHAINing and CHAINed programs. With Microsoft BASIC Compiler, the best way to insure this is to place all blank COMMON declarations in a single include file and use the $INCLUDE statement in each program.

For example:

```
MENU.BAS
        10 $INCLUDE COMDEF
         .
         .
         .
        1000 CHAIN "PROG1"

PROG1.BAS
        10 $INCLUDE COMDEF
         .
         .
         .
        2000 CHAIN "MENU"

COMDEF.BAS
        100 DIM A(100),B$(200)
        110 COMMON I,J,K,A()
        120 COMMON A$,B$(),X,Y,Z
```

2.8 CONT

Syntax CONT

Purpose To continue program execution after a Control-C has been typed or a STOP or END statement has been executed.

Remarks Execution resumes at the point where the break occurred. If the break occurred after a prompt from an INPUT statement, execution continues with the reprinting of the prompt ("?" or prompt string).

CONT is usually used in conjunction with STOP for debugging. When execution is stopped, intermediate values may be examined and changed using direct mode statements. Execution may be resumed with CONT or a direct mode GOTO, which resumes execution at a specified line number. CONT may be used to continue execution after an error has occurred.

CONT is invalid if the program has been edited during the break.

Example See "STOP," Section 2.61.

2.9 CSAVE

Syntax CSAVE <string expression>
CSAVE* <array variable name>

Purpose To save the program or an array currently in memory on cassette tape.

Remarks

Each program or array saved on tape is identified by a filename. When the command CSAVE <string expression> is executed, Microsoft BASIC saves the program currently in memory on tape and uses the first character in <string expression> as the filename. <string expression> may be more than one character, but only the first character is used for the filename.

When the command CSAVE* <array variable name> is executed, Microsoft BASIC saves the specified array on tape. The array must be a numeric array. The elements of a multidimensional array are saved with the leftmost subscript changing fastest. For example, when the 2-dimensional array specified by DIM A(2,2) is saved (see "DIM," Section 2.15), the array elements are saved in this order:

0,0
1,0
2,0
0,1
1,1
2,1
0,2
1,2
2,2

CSAVE may be used as a program statement or as a direct mode command.

Before a CSAVE or CSAVE* is executed, make sure the cassette recorder is properly connected and in the record mode.

See also "CLOAD," Section 2.5.

Note

CSAVE and CLOAD are not included in all implementations of Microsoft BASIC.

Example

```
CSAVE "TIMER"
```

Saves the program currently in memory on cassette under filename "TIMER".

2.10 DATA

Syntax DATA <list of constants>

Purpose To store the numeric and string constants that are accessed by the program's READ statement(s). (See "READ," Section 2.54.)

Remarks DATA statements are nonexecutable and may be placed anywhere in the program. A DATA statement may contain as many constants as will fit on a line (separated by commas). Any number of DATA statements may be used in a program. READ statements access DATA statements in order (by line number). The data contained therein may be thought of as one continuous list of items, regardless of how many items are on a line or where the lines are placed in the program.

<list of constants> may contain numeric constants in any format, i.e., fixed-point, floating-point, or integer. (No numeric expressions are allowed in the list.) String constants in DATA statements must be surrounded by double quotation marks only if they contain commas, colons, or significant leading or trailing spaces. Otherwise, quotation marks are not needed.

The variable type (numeric or string) given in the READ statement must agree with the corresponding constant in the DATA statement.

DATA statements may be reread from the beginning by use of the RESTORE statement (Section 2.57).

Example See "READ," Section 2.54.

2.11 DEF FN

Syntax

DEF FN<name>[(<parameter list>)] =
<function definition>

Purpose

To define and name a function that is written by the user.

Remarks

<name> must be a legal variable name. This name, preceded by FN, becomes the name of the function.

<parameter list> is comprised of those variable names in the function definition that are to be replaced when the function is called. The items in the list are separated by commas.

<function definition> is an expression that performs the operation of the function. It is limited to one line. Variable names that appear in this expression serve only to define the function; they do not affect program variables that have the same name. A variable name used in a function definition may or may not appear in the parameter list. If it does, the value of the parameter is supplied when the function is called. Otherwise, the current value of the variable is used.

The variables in the parameter list represent, on a one-to-one basis, the argument variables or values that will be given in the function call.

This statement may define either numeric or string functions. If a type is specified in the function name, the value of the expression is forced to that type before it is returned to the calling statement. If a type is specified in the function name and the argument type does not match, a "Type mismatch" error occurs.

A DEF FN statement must be executed before the function it defines may be called. If a function is called before it has been defined, an "Undefined user function" error occurs. DEF FN is illegal in the direct mode.

Example

```
410 DEF FNAB(X,Y) = X^3/Y^2
420 T = FNAB(I,J)
```

Line 410 defines the function FNAB. The function is called in line 420.

2.12 DEFINT/SNG/DBL/STR

Syntax DEF<type> <range(s) of letters>

where <type> is INT, SNG, DBL, or STR

Purpose To declare variable types as integer, single precision, double precision, or string.

Remarks Any variable names beginning with the letter(s) specified in <range of letters> will be considered the type of variable specified in the <type> portion of the statement. However, a type declaration character always takes precedence over a DEF-type statement. (See "Variable Names and Declaration Characters," Section 1.6.1.)

If no type declaration statements are encountered, Microsoft BASIC assumes all variables without declaration characters are single precision variables.

Examples

10 DEFDBL L-P	All variables beginning with the letters L, M, N, O, and P will be double precision variables.
10 DEFSTR A	All variables beginning with the letter A will be string variables.
10 DEFINT I-N, W-Z	All variables beginning with the letters I, J, K, L, M, N, W, X, Y, Z will be integer variables.

2.13 DEF USR

Syntax

DEF USR[<digit>] = <integer expression>

Purpose

To specify the starting address of an assembly language subroutine.

Remarks

<digit> may be any digit from 0 to 9. The digit corresponds to the number of the USR routine whose address is being specified. If <digit> is omitted, DEF USR0 is assumed. The value of <integer expression> is the starting address of the USR routine. See "Assembly Language Subroutines," in the *Microsoft BASIC User's Guide.*

Any number of DEF USR statements may appear in a program to redefine subroutine starting addresses, thus allowing access to as many subroutines as necessary.

Example

```
.
.
.
200 DEF USR0 = 24000
210 X = USR0(Y^2/2.89)
.
.
.
```

2.14 DELETE

Syntax DELETE [<line number>][-<line number>]

Purpose To delete program lines.

Remarks Microsoft BASIC always returns to command level after a DELETE is executed. If <line number> does not exist, an "Illegal function call" error occurs.

Examples

DELETE 40	Deletes line 40.
DELETE 40-100	Deletes lines 40 through 100, inclusive.
DELETE -40	Deletes all lines up to and including line 40.

2.15 DIM

Syntax DIM <list of subscripted variables>

Purpose To specify the maximum values for array variable subscripts and allocate storage accordingly.

Remarks If an array variable name is used without a DIM statement, the maximum value of the array's subscript(s) is assumed to be 10. If a subscript is used that is greater than the maximum specified, a "Subscript out of range" error occurs. The minimum value for a subscript is always 0, unless otherwise specified with the OPTION BASE statement (see Section 2.46).

The DIM statement sets all the elements of the specified arrays to an initial value of zero.

Example

```
10 DIM A(20)
20 FOR I = 0 TO 20
30 READ A(I)
40 NEXT I
 .
 .
 .
```

2.16 EDIT

Syntax EDIT <line number>

Purpose To enter edit mode at the specified line.

Remarks In edit mode, it is possible to edit portions of a line without retyping the entire line. Upon entering edit mode, BASIC types the line number of the line to be edited, then it types a space and waits for an edit mode subcommand.

Edit Mode Subcommands

Edit mode subcommands are used to move the cursor or to insert, delete, replace, or search for text within a line. The subcommands are not echoed. However, most of the edit mode subcommands may be preceded by an integer which causes the command to be executed that number of times. When an integer is not specified, it is assumed to be 1.

Edit mode subcommands may be categorized according to the following functions:

1. Moving the cursor
2. Inserting text
3. Deleting text
4. Finding text
5. Replacing text

6. Ending and restarting edit mode
7. Entering edit mode from a syntax error

Note

In the descriptions that follow, <ch> represents any character, <text> represents a string of characters of arbitrary length, [i] represents an optional integer (the default is 1), and $ represents the Escape (or Altmode) key.

---

1. Moving the Cursor

   Space bar — Use the space bar to move the cursor to the right. [i]Space bar moves the cursor i spaces to the right. Characters are printed as you space over them.

   Rubout — In edit mode, [i]Rubout moves the cursor i spaces to the left (backspaces). Characters are printed as you backspace over them.

2. Inserting Text

   I — I<text>$ inserts <text> at the current cursor position. The inserted characters are printed on the terminal. To terminate insertion, press Escape. If a <carriage return> is typed during an Insert command, the effect is the same as pressing Escape and then <carriage return>. During an Insert command, the Rubout, Delete, or Underscore key on the terminal may be used to delete characters to the left of the cursor. Rubout will print out the characters as you backspace over them. Delete and Underscore will print an Underscore for each character that you backspace over. If an attempt is made to insert a character that will make the line longer than 255 characters, a bell (Control-G) sounds and the character is not printed.

X — The X subcommand extends the line. X moves the cursor to the end of the line, enters insert mode, and allows insertion of text as if an Insert command had been given. When you are finished extending the line, press Escape or carriage return.

3. Deleting Text

D — [i]D deletes i characters to the right of the cursor. The deleted characters are echoed between backslashes, and the cursor is positioned to the right of the last character deleted. If there are fewer than i characters to the right of the cursor, iD deletes the remainder of the line.

H — H deletes all characters to the right of the cursor and then automatically enters insert mode. H is useful for extending a line or replacing statements at the end of a line.

4. Finding Text

S — The subcommand [i]S<ch> searches for the ith occurrence of <ch> and positions the cursor before it. The character at the current cursor position is not included in the search. If <ch> is not found, the cursor stops at the end of the line. All characters passed over during the search are printed.

K — The subcommand [i]K<ch> is similar to [i]S<ch>, except all the characters passed over in the search are deleted. The cursor is positioned before <ch>, and the deleted characters are enclosed in backslashes.

5. Replacing Text

C — The subcommand C<ch> changes the next character to <ch>. If you wish to change the next i characters, use the subcommand iC, followed by as many characters as are specified by i. After the ith new character is typed, change mode is exited and you will return to edit mode.

6. Ending and Restarting Edit Mode

<cr>	Typing a <carriage return> prints the remainder of the line, saves the changes you made, and exits edit mode.
E	The E subcommand has the same effect as <carriage return>, except the remainder of the line is not printed.
Q	The Q subcommand returns to Microsoft BASIC command level, *without* saving any of the changes that were made to the line in edit mode.
L	The L subcommand lists the remainder of the line (saving any changes made so far) and repositions the cursor at the beginning of the line, still in edit mode. L is usually used to list the line when you first enter edit mode.
A	The A subcommand lets you begin editing a line over again. It restores the original line and repositions the cursor at the beginning.
Control-A	To enter edit mode on the line you are currently typing, type Control-A. Microsoft BASIC responds with a <carriage return>, an exclamation point (!), and a space. The cursor will be positioned at the first character in the line. Proceed by typing an edit mode subcommand.

Remember, if you have just entered a line and wish to go back and edit it, the command "EDIT." will enter edit mode at the current line. (The line number symbol "." always refers to the current line.)

If an unrecognizable command or illegal character is input to Microsoft BASIC while in edit mode, BASIC sends a Control-G (bell) to the terminal and the command or character is ignored.

7. Entering Edit Mode from a Syntax Error

When a syntax error is encountered during execution of a program, Microsoft BASIC automatically enters edit mode at the line that caused the error. For example:

```
10 K = 2(4)
RUN
?Syntax error in 10
10
```

When you finish editing the line and press <carriage return> (or the E subcommand), Microsoft BASIC reinserts the line. This causes all variable values to be lost. To preserve the variable values for examination, first exit edit mode with the Q subcommand. Microsoft BASIC will return to command level, and all variable values will be preserved.

2.17 END

Syntax END

Purpose To terminate program execution, close all files, and return to command level.

Remarks END statements may be placed anywhere in the program to terminate execution. Unlike the STOP statement, END does not cause a "Break in line nnnnn" message to be printed. An END statement at the end of a program is optional. Microsoft BASIC always returns to command level after an END is executed.

Example

```
520 IF K>1000 THEN END ELSE GOTO 20
```

2.18 ERASE

Syntax ERASE <list of array variables>

Purpose To eliminate arrays from a program.

Remarks Arrays may be redimensioned after they are ERASEd, or the previously allocated array space in memory may be used for other purposes. If an attempt is made to redimension an array without first ERASEing it, a "Redimensioned array" error occurs.

Microsoft BASIC Compiler does not support ERASE.

Example

```
.
.
.
450 ERASE A,B
460 DIM B(99)
.
.
.
```

2.19 ERR and ERL Variables

When an error handling routine is entered, the variable ERR contains the error code for the error and the variable ERL contains the line number of the line in which the error was detected. The ERR and ERL variables are usually used in IF...THEN statements to direct program flow in the error handling routine.

If the statement that caused the error was a direct mode statement, ERL will contain 65535. To test whether an error occurred in a direct statement, use IF 65535=ERL THEN .... Otherwise, use

```
IF ERR = error code THEN ...
IF ERL = line number THEN ...
```

If the line number is not on the right side of the relational operator, it cannot be renumbered with RENUM. Because ERL and ERR are reserved variables, neither may appear to the left of the equal sign in a LET (assignment) statement. Microsoft BASIC error codes are listed in Appendix A.

2.20 ERROR

Syntax ERROR <integer expression>

Purpose To simulate the occurrence of a BASIC error, or to allow 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 of <integer expression> equals an error code already in use by BASIC (see Appendix A), the ERROR statement will simulate the occurrence of that error and the corresponding error message will be printed. (See Example 1.)

To define your own error code, use a value that is greater than any used by MS-BASIC error codes. (It is preferable to use the highest available values, so compatibility may be maintained when more error codes are added to MS-BASIC.) This user-defined error code may then be conveniently handled in an error handling routine. (See Example 2.)

If an ERROR statement specifies a code for which no error message has been defined, MS-BASIC responds with the "Unprintable error" error message. Execution of an ERROR statement for which there is no error handling routine causes an error message to be printed and execution to halt.

Example 1

```
LIST
10 S = 10
20 T = 5
30 ERROR S + T
40 END
Ok
RUN
String too long in line 30
```

Or, in direct mode:

```
Ok
ERROR 15                    (You type this line.)
String too long             (BASIC types this line.)
Ok
```

Example 2

```
.
.
.
110 ON ERROR GOTO 400
120 INPUT "WHAT IS YOUR BET";B
130 IF B>5000 THEN ERROR 210
.
.
.
400 IF ERR = 210 THEN PRINT "HOUSE LIMIT IS
$5000"
410 IF ERL = 130 THEN RESUME 120
.
.
.
```

2.21 FIELD

Syntax

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

Purpose

To allocate space for variables in a random file buffer.

Remarks

Before a GET statement or PUT statement can be executed, a FIELD statement must be executed to format the random file buffer.

<file number> is the number under which the file was OPENed. <field width> is the number of characters to be allocated to <string variable>. For example,

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

allocates the first 20 positions (bytes) in the random file buffer to the string variable N$, the next 10 positions to ID$, and the next 40 positions to ADD$. FIELD does not place any data in the random file buffer. (See "LSET and RSET," Section 2.37, and "GET," Section 2.23.)

The total number of bytes allocated in a FIELD statement must not exceed the record length that was specified when the file was OPENed. Otherwise, a "Field overflow" error occurs. (The default record length is 128 bytes.)

Any number of FIELD statements may be executed for the same file. All FIELD statements that have been executed will remain in effect at the same time.

Note

Do not use a FIELDed variable name in an INPUT or LET statement. Once a variable name is FIELDed, it points to the correct place in the ran-

dom file buffer. If a subsequent INPUT or LET statement with that variable name is executed, the variable's pointer is moved to string space.

Example 1

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

Allocates the first 20 positions (bytes) in the random file buffer to the string variable N$, the next 10 positions to ID$, and the next 40 positions to ADD$. FIELD does NOT place any data in the random file buffer. (See also "GET," Section 2.23, and "LSET and RSET," Section 2.37.)

Example 2

```
10 OPEN "R,"#1,"A:PHONELST",35
15 FIELD #1,2 AS RECNBR$,33 AS DUMMY$
20 FIELD #1,25 AS NAMES,10 AS PHONENBR$
25 GET #1
30 TOTAL = CVI(RECNBR)$
35 FOR I = 2 TO TOTAL
40 GET #1, I
45 PRINT NAMES, PHONENBR$
50 NEXT I
```

Illustrates a multiple defined FIELD statement. In statement 15, the 35-byte field is defined for the first record to keep track of the number of records in the file. In the next loop of statements (35–50), statement 20 defines the field for individual names and phone numbers.

Example 3

```
10 FOR LOOP% = 0 TO 7
20 FIELD #1,(LOOP% * 16) AS OFFSETS,16 AS
A$(LOOP%)
30 NEXT LOOP%
```

Shows the construction of a FIELD statement using an array of elements of equal size. The result is equivalent to the single declaration:

```
FIELD#1,16 AS A$(0),16 AS A$(1),. . .,16 AS A$(6),16
AS A$(7)
```

Example 4

```
10 DIM SIZE% (NUMB%): REM ARRAY OF FIELD
SIZES
20 FOR LOOP% = 0 TO NUMB%:READ SIZE%
(LOOP%): NEXT LOOP%
30 DATA 9,10,12,21,41
  .
  .
  .
120 DIM A$(NUMB%): REM ARRAY OF FIELDED
VARIABLES
130 OFFSET% = 0
140 FOR LOOP% = 0 TO NUMB%
150 FIELD #1, OFFSET% AS OFFSET$, SIZE%
(LOOP%) AS A$(LOOP%)
160 OFFSET% = OFFSET% + SIZE%(LOOP%)
170 NEXT LOOP%
```

Creates a field in the same manner as Example 3. However, the element size varies with each element. The equivalent declaration is:

```
FIELD #1,SIZE%(0) AS A$(0),SIZE%(1) AS A$(1),. . .
SIZE%(NUMB%) AS A$(NUMB%)
```

2.22 FOR...NEXT

Syntax

```
FOR <variable> = x TO y [STEP z]
  .
  .
  .
NEXT [<variable>][,<variable>...]
```

where x, y, and z are numeric expressions.

Purpose

To allow a series of instructions to be performed in a loop a given number of times.

Remarks

<variable> is used as a counter. The first numeric expression (x) is the initial value of the counter. The second numeric expression (y) is the final value of the counter. The program lines following

the FOR statement are executed until the NEXT statement is encountered. Then the counter is adjusted by the amount specified by STEP. A check is performed to see if the value of the counter is now greater than the final value (y). If it is not greater, Microsoft BASIC branches back to the statement after the FOR statement and the process is repeated. If it is greater, execution continues with the statement following the NEXT statement. This is a FOR...NEXT loop.

If STEP is not specified, the increment is assumed to be one. If STEP is negative, the final value of the counter is set to be less than the initial value. The counter is decreased each time through the loop. The loop is executed until the counter is less than the final value.

The counter must be an integer or single precision numeric constant. If a double precision numeric constant is used, a "Type mismatch" error will result.

The body of the loop is skipped if the initial value of the loop times the sign of the STEP exceeds the final value times the sign of the STEP.

Nested Loops

FOR...NEXT loops may be nested; that is, a FOR...NEXT loop may be placed within the context of another FOR...NEXT loop. When loops are nested, each loop must have a unique variable name as its counter. The NEXT statement for the inside loop must appear before that for the outside loop. If nested loops have the same end point, a single NEXT statement may be used for all of them.

The variable(s) in the NEXT statement may be omitted, in which case the NEXT statement will match the most recent FOR statement. If a NEXT statement is encountered before its corresponding FOR statement, a "NEXT without FOR" error message is issued and execution is terminated.

Example 1

```
10 K = 10
20 FOR I = 1 TO K STEP 2
30 PRINT I;
40 K = K + 10
50 PRINT K
60 NEXT
RUN
 1   20
 3   30
 5   40
 7   50
 9   60
Ok
```

Example 2

```
10 J = 0
20 FOR I = 1 TO J
30 PRINT I
40 NEXT I
```

In this example, the loop does not execute because the initial value of the loop exceeds the final value.

Example 3

```
10 I = 5
20 FOR I = 1 TO I + 5
30 PRINT I;
40 NEXT
RUN
 1  2  3  4  5  6  7  8  9  10
Ok
```

In this example, the loop executes ten times. The final value for the loop variable is always set before the initial value is set.

Note

Previous versions of Microsoft BASIC set the initial value of the loop variable before setting the final value; i.e., the above loop would have executed six times.

2.23 GET

Syntax GET [#]<file number>[,<record number>]

Purpose To read a record from a random disk file into a random buffer.

Remarks <file number> is the number under which the file was OPENed. If <record number> is omitted, the next record (after the last GET) is read into the buffer. The largest possible record number is 32767.

Example See "Disk File Handling," in the *Microsoft BASIC User's Guide.*

Note After a GET statement has been executed, INPUT# and LINE INPUT# may be executed to read characters from the random file buffer.

2.24 GOSUB...RETURN

Syntax

```
GOSUB <line number>
.
.
.
RETURN
```

Purpose To branch to and return from a subroutine.

Remarks <line number> is the first line of the subroutine.

A subroutine may be called any number of times in a program. A subroutine also may be called from within another subroutine. Such nesting of subroutines is limited only by available memory.

The RETURN statement(s) in a subroutine cause Microsoft BASIC to branch back to the statement following the most recent GOSUB statement. A subroutine may contain more than one RETURN statement, should logic dictate a return at different points in the subroutine. Subroutines may appear anywhere in the program, but it is recommended that the subroutine be readily distinguishable from the main program. To prevent inadvertent entry into the subroutine, precede it with a STOP, END, or GOTO statement that directs program control around the subroutine.

Example

```
10 GOSUB 40
20 PRINT "BACK FROM SUBROUTINE"
30 END
40 PRINT "SUBROUTINE";
50 PRINT " IN";
60 PRINT " PROGRESS"
70 RETURN
RUN
SUBROUTINE IN PROGRESS
BACK FROM SUBROUTINE
Ok
```

2.25 GOTO

Syntax

GOTO <line number>

Purpose

To branch unconditionally out of the normal program sequence to a specified line number.

Remarks

If <line number> is an executable statement, that statement and those following are executed. If it is a nonexecutable statement, execution proceeds at the first executable statement encountered after <line number>.

Example

```
LIST
10 READ R
20 PRINT "R = ";R,
30 A = 3.14*R^2
40 PRINT "AREA = ";A
50 GOTO 10
60 DATA 5,7,12
Ok
RUN
R = 5       AREA = 78.5
R = 7       AREA = 153.86
R = 12      AREA = 452.16
?Out of data in 10
Ok
```

2.26 IF...THEN[...ELSE] and IF...GOTO

Syntax 1

IF <expression> THEN {<statement(s)>|<line number>} [ELSE {<statement(s)>|<line number>}]

Syntax 2

IF <expression> GOTO <line number> [ELSE {<statement(s)>|<line number>}]

Purpose

To make a decision regarding program flow based on the result returned by an expression.

Remarks

If the result of <expression> is not zero, the THEN or GOTO clause is executed. THEN may be followed by either a line number for branching or one or more statements to be executed. GOTO is always followed by a line number. If the result of <expression> is zero, the THEN or GOTO clause is ignored and the ELSE clause, if present, is executed. Execution continues with the next executable statement. A comma is allowed before THEN.

Nesting of IF Statements

IF...THEN...ELSE statements may be nested. Nesting is limited only by the length of the line.

For example,

```
IF X>Y THEN PRINT "GREATER" ELSE IF Y>X
THEN PRINT "LESS THAN" ELSE PRINT "EQUAL"
```

is a legal statement. If the statement does not contain the same number of ELSE and THEN clauses, each ELSE is matched with the closest unmatched THEN. For example,

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

will not print "A<>C" when A<>B.

If an IF...THEN statement is followed by a line number in direct mode, an "Undefined line" error results, unless a statement with the specified line number had previously been entered in indirect mode.

Note

When using IF to test equality for a value that is the result of a floating-point computation, remember that the internal representation of the value may not be exact. Therefore, the test should be against the range over which the accuracy of the value may vary. For example, to test a computed variable A against the value 1.0, use:

```
IF ABS (A-1.0)<1.0E-6 THEN ...
```

This test returns true if the value of A is 1.0 with a relative error of less than 1.0E-6.

Example 1

```
200 IF I THEN GET#1,I
```

This statement GETs record number I if I is not zero.

Example 2

```
100 IF(I<20)*(I>10) THEN DB = 1979-1:GOTO 300
110 PRINT "OUT OF RANGE"
.
.
.
```

In this example, a test determines if I is greater than 10 and less than 20. If I is in this range, DB is calculated and execution branches to line 300. If I is not in this range, execution continues with line 110.

Example 3

```
210 IF IOFLAG THEN PRINT A$ ELSE LPRINT A$
```

This statement causes printed output to go either to the terminal or the line printer, depending on the value of the variable IOFLAG. If IOFLAG is zero, output goes to the line printer; otherwise, output goes to the terminal.

2.27 INPUT

Syntax

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

Purpose

To allow input from the terminal during program execution.

Remarks

When an INPUT statement is encountered, program execution pauses and a question mark is printed to indicate the program is waiting for data. If <"prompt string"> is included, the string is printed before the question mark. The required data is then entered at the terminal.

A comma may be used instead of a semicolon after the prompt string to suppress the question mark. For example, the statement INPUT "ENTER BIRTHDATE",B$ will print the prompt with no question mark.

If INPUT is immediately followed by a semicolon, then the carriage return typed to input data does not echo a carriage return/line feed sequence.

The data that is entered is assigned to the variable(s) given in <variable list>. The number of data items supplied must be the same as the number of variables in the list. Data items are separated by commas.

The variable names in the list may be numeric or string variable names (including subscripted variables). The type of each data item that is input must agree with the type specified by the variable name. (Strings input to an INPUT statement need not be surrounded by quotation marks.)

Responding to INPUT with too many or too few items or with the wrong type of value (numeric instead of string, etc.) causes the message "?Redo from start" to be printed. No assignment of input values is made until an acceptable response is given.

Examples

```
10 INPUT X
20 PRINT X "SQUARED IS" X^2
30 END
RUN
? 5                          (The 5 was typed in by the
                              user in response to the ques-
                              tion mark.)
  5 SQUARED IS 25
Ok

LIST
10 PI = 3.14
20 INPUT "WHAT IS THE RADIUS";R
30 A = PI*R^2
40 PRINT "THE AREA OF THE CIRCLE? IS";A
50 PRINT
60 GOTO 20
Ok
RUN
WHAT IS THE RADIUS? 7.4      (User types 7.4)
THE AREA OF THE CIRCLE IS 171.946

WHAT IS THE RADIUS?
etc.
```

2.28 INPUT#

Syntax INPUT#<file number>,<variable list>

Purpose To read data items from a sequential disk file and assign them to program variables.

Remarks <file number> is the number used when the file was OPENed for input. <variable list> contains the variable names that will be assigned to the items in the file. (The variable type must match the type specified by the variable name.) With INPUT#, no question mark is printed, as with INPUT.

The data items in the file should appear just as they would if data were being typed in response to an INPUT statement. With numeric values, leading spaces, carriage returns, and line feeds are ignored. The first character encountered that is not a space, carriage return, or line feed is assumed to be the start of a number. The number terminates on a space, carriage return, line feed, or comma.

If Microsoft BASIC is scanning the sequential data file for a string item, leading spaces, carriage returns, and line feeds are also ignored. The first character encountered that is not a space, carriage return, or line feed is assumed to be the start of a string item. If this first character is a quotation mark ("), the string item will consist of all characters read between the first quotation mark and the second. Thus, a quoted string may not contain a quotation mark as a character. If the first character of the string is not a quotation mark, the string is an unquoted string, and will terminate on a comma, carriage return, or line feed (or after 255 characters have been read). If end-of-file is reached when a numeric or string item is being INPUT, the item is terminated.

Example See "Disk File Handling," in the *Microsoft BASIC User's Guide.*

2.29 KILL

Syntax KILL <filename>

Purpose To delete a file from disk.

Remarks If a KILL statement is given for a file that is currently OPEN, a "File already open" error occurs.

KILL is used for all types of disk files: program files, random data files, and sequential data files.

Example

```
200 KILL "DATA1.DAT"
```

See also "Disk File Handling," in the *Microsoft BASIC User's Guide.*

2.30 LET

Syntax [LET]<variable> = <expression>

Purpose To assign the value of an expression to a variable.

Remarks Notice the word LET is optional; i.e., the equal sign is sufficient for assigning an expression to a variable name.

Example

```
110 LET D = 12
120 LET E = 12^2
130 LET F = 12^4
140 LET SUM = D + E + F
 .
 .
 .
```

or

```
110 D = 12
120 E = 12^2
130 F = 12^4
140 SUM = D + E + F
 .
 .
 .
```

2.31 LINE INPUT

Syntax

LINE INPUT[;] [<"prompt string">;]
<string variable>

Purpose

To input an entire line (up to 254 characters) to a string variable, without the use of delimiters.

Remarks

<"prompt string"> is a string literal that is printed at the terminal before input is accepted. A question mark is not printed unless it is part of <"prompt string">. All input from the end of <"prompt string"> to the carriage return is assigned to <string variable>. However, if a line feed/carriage return sequence (this order only) is encountered, both characters are echoed; but the carriage return is ignored, the line feed is put into <string variable>, and data input continues.

If LINE INPUT is immediately followed by a semicolon, then the carriage return typed by the user to end the input line does not echo a carriage return/line feed sequence at the terminal.

A LINE INPUT statement may be aborted by typing Control-C. Microsoft BASIC will return to command level and type "Ok". Typing CONT resumes execution at the LINE INPUT.

Example See "LINE INPUT#," Section 2.32.

2.32 LINE INPUT#

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

Purpose To read an entire line (up to 254 characters), without delimiters, from a sequential disk data file to a string variable.

Remarks <file number> is the number under which the file was OPENed. <string variable> is the variable name to which the line will be assigned. LINE INPUT# reads all characters in the sequential file up to a carriage return. It then skips over the carriage return/line feed sequence. The next LINE INPUT# reads all characters up to the next carriage return. (If a line feed/carriage return sequence is encountered, it is preserved.)

LINE INPUT# is especially useful if each line of a data file has been broken into fields, or if a Microsoft BASIC program saved in ASCII format is being read as data by another program. (See "SAVE," Section 2.60.)

Example

```
10 OPEN "0",1,"LIST"
20 LINE INPUT "CUSTOMER INFORMATION?";C$
30 PRINT #1, C$
40 CLOSE 1
50 OPEN "I",1,"LIST"
60 LINE INPUT #1, C$
70 PRINT C$
80 CLOSE 1
RUN
CUSTOMER INFORMATION? LINDA JONES  234,4
MEMPHIS
LINDA JONES  234,4  MEMPHIS
Ok
```

2.33 LIST

Syntax 1 LIST[<line number>]

Syntax 2 LIST [<line number>][-[<line number>]]

Purpose To list all or part of the program currently in memory at the terminal.

Remarks Microsoft BASIC always returns to command level after a LIST is executed.

Syntax 1

If <line number> is omitted, the program is listed beginning at the lowest line number. (Listing is terminated either when the end of the program is reached or by typing Control-C.) If <line number> is included, only the specified line will be listed.

Syntax 2

This syntax allows the following options:

1. If only the first <line number> is specified, that line and all higher-numbered lines are listed.

2. If only the second <line number> (i.e., [-[<line number>]]) is specified, all lines from the beginning of the program through that line are listed.

3. If both <line number(s)> are specified, the entire range is listed.

Examples

Syntax 1

LIST	Lists the program currently in memory.
LIST 500	Lists line 500.

Syntax 2

LIST 150-	Lists all lines from 150 to the end.
LIST -1000	Lists all lines from the lowest number through 1000.
LIST 150-1000	Lists lines 150 through 1000, inclusive.

2.34 LLIST

Syntax LLIST [<line number>[-[<line number>]]]

Purpose To list all or part of the program currently in memory at the line printer.

Remarks LLIST assumes a 132-character-wide printer.

Microsoft BASIC always returns to command level after an LLIST is executed. The options for LLIST are the same as for LIST, Syntax 2.

Note LLIST and LPRINT are not included in all implementations of Microsoft BASIC.

Example See the examples for "LIST," Syntax 2.

2.35 LOAD

Syntax LOAD <filename>[,R]

Purpose To load a file from disk into memory.

Remarks <filename> is the name that was used when the file was SAVEd. (Your operating system may append a default filename extension if one was not supplied in the SAVE command. Refer to "Microsoft BASIC Disk I/O," in the *Microsoft BASIC User's Guide,* for information about possible filename extensions under your operating system.)

The R option automatically runs the program after it has been loaded.

LOAD closes all open files and deletes all variables and program lines currently residing in memory before it loads the designated program. However, if the R option is used with LOAD, the program is RUN after it is LOADed, and all open data files are kept open. Thus, LOAD with the R option may be used to chain several programs (or segments of the same program). Information may be passed between the programs using their disk data files.

Example

```
LOAD "STRTRK",R
LOAD "B:MYPROG"
```

2.36 LPRINT and LPRINT USING

Syntax

```
LPRINT [<list of expressions>]
LPRINT USING <string exp>;<list of expressions>
```

Purpose To print data at the line printer.

Remarks Same as PRINT and PRINT USING, except output goes to the line printer. See Section 2.49 and Section 2.50.

LPRINT assumes a 132-character-wide printer.

Note LPRINT and LLIST are not included in all implementations of Microsoft BASIC.

2.37 LSET and RSET

Syntax

LSET <string variable> = <string expression>
RSET <string variable> = <string expression>

Purpose To move data from memory to a random file buffer (in preparation for a PUT statement).

Remarks If <string expression> requires fewer bytes than were FIELDed to <string variable>, LSET left-justifies the string in the field, and RSET right-justifies the string. (Spaces are used to pad the extra positions.) If the string is too long for the field, characters are dropped from the right. Numeric values must be converted to strings before they are LSET or RSET. See "MKI$, MKS$, MKD$," Section 3.26.

Examples

```
150 LSET A$ = MKS$(AMT)
160 LSET D$ = DESC($)
```

See also "Disk File Handling," in the *Microsoft BASIC User's Guide.*

Note LSET or RSET may also be used with a nonfielded string variable to left-justify or right-justify a string in a given field. For example, the program lines

```
110 A$ = SPACE$(20)
120 RSET A$ = N$
```

right-justify the string N$ in a 20-character field. This can be very handy for formatting printed output.

2.38 MERGE

Syntax MERGE <filename>

Purpose To merge a specified disk file into the program currently in memory.

Remarks <filename> is the name used when the file was SAVEd. (Your operating system may append a default filename extension if one was not supplied in the SAVE command. Refer to "Disk File Handling," in the *Microsoft BASIC User's Guide*, for information about possible filename extensions under your operating system.) The file must have been SAVEd in ASCII format. (If not, a "Bad file mode" error occurs.)

If any lines in the disk file have the same line numbers as lines in the program in memory, the lines from the file on disk will replace the corresponding lines in memory. (MERGEing may be thought of as "inserting" the program lines on disk into the program in memory.)

Microsoft BASIC always returns to command level after executing a MERGE command.

Example MERGE "NUMBRS"

2.39 MID$

Syntax MID$(<string exp1>,n[,m])=<string exp2>

where n and m are integer expressions and <string exp1> and <string exp2> are string expressions.

Purpose To replace a portion of one string with another string.

Remarks The characters in <string exp1>, beginning at position n, are replaced by the characters in <string exp2>. The optional "m" refers to the number of characters from <string exp2> that will be used in the replacement. If "m" is omitted, all of <string exp2> is used. However, regardless of whether "m" is omitted or included, the replacement of characters never goes beyond the original length of <string exp1>.

Example

```
10 A$ = "KANSAS CITY, MO"
20 MID$(A$,14) = "KS"
30 PRINT A$
RUN
KANSAS CITY, KS
```

MID$ is also a function that returns a substring of a given string. See Section 3.25.

2.40 NAME

Syntax NAME <old filename> AS <new filename>

Purpose To change the name of a disk file.

Remarks <old filename> must exist and <new filename> must not exist; otherwise, an error will result. After a NAME command, the file exists on the same disk, in the same area of disk space, with the new name.

Example

```
Ok
NAME "ACCTS" AS "LEDGER"
Ok
```

In this example, the file that was formerly named ACCTS will now be named LEDGER.

2.41 NEW

Syntax NEW

Purpose To delete the program currently in memory and clear all variables.

Remarks NEW is entered at command level to clear memory before entering a new program. Microsoft BASIC always returns to command level after a NEW is executed.

Example

```
NEW
```

2.42 NULL

Syntax NULL <integer expression>

Purpose To set the number of nulls to be printed at the end of each line.

Remarks For 10 character-per-second tape punches, <integer expression> should be >=3. When tapes are not being punched, <integer expression> should be 0 or 1 for Teletype® and Teletype-compatible terminal screens. <integer expression> should be 2 or 3 for 30 CPS hard copy printers. The default value is 0.

Example

```
Ok
NULL 2
Ok
100 INPUT X
200 IF X<50 GOTO 800
.
.
.
```

Two null characters will be printed after each line.

2.43 ON ERROR GOTO

Syntax

ON ERROR GOTO <line number>

Purpose

To enable error handling and specify the first line of the error handling routine.

Remarks

Once error handling has been enabled, all errors detected, including direct mode errors (e.g., syntax errors), will cause a jump to the specified error handling routine. If <line number> does not exist, an "Undefined line" error results.

To disable error handling, execute an ON ERROR GOTO 0. Subsequent errors will print an error message and halt execution. An ON ERROR GOTO 0 statement that appears in an error handling routine causes Microsoft BASIC to stop and print the error message for the error that caused the trap. It is recommended that all error handling routines execute an ON ERROR GOTO 0 if an error is encountered for which there is no recovery action.

Note

If an error occurs during execution of an error handling routine, that error message is printed and execution terminates. Error trapping does not occur within the error handling routine.

Example

```
10 ON ERROR GOTO 1000
```

2.44 ON...GOSUB and ON...GOTO

Syntax

ON <expression> GOTO <list of line numbers>
ON <expression> GOSUB <list of line numbers>

Purpose

To branch to one of several specified line numbers, depending on the value returned when an expression is evaluated.

Remarks The value of <expression> determines which line number in the list will be used for branching. For example, if the value is three, the third line number in the list will be the destination of the branch. (If the value is a noninteger, the fractional portion is rounded.)

In the ON...GOSUB statement, each line number in the list must be the first line number of a subroutine.

If the value of <expression> is zero or greater than the number of items in the list (but less than or equal to 255), Microsoft BASIC continues with the next executable statement. If the value of <expression> is negative or greater than 255, an "Illegal function call" error occurs.

Example

```
100 ON L-1 GOTO 150,300,320,390
```

2.45 OPEN

Syntax OPEN <mode>,[#]<file number>,<filename>[,<reclen>]

Purpose To allow I/O to a disk file.

Remarks A disk file must be OPENed before any disk I/O operation can be performed on that file. OPEN allocates a buffer for I/O to the file and determines the mode of access that will be used with the buffer.

<mode> is a string expression whose first character is one of the following:

O Specifies sequential output mode.

I Specifies sequential input mode.

R Specifies random input/output mode.