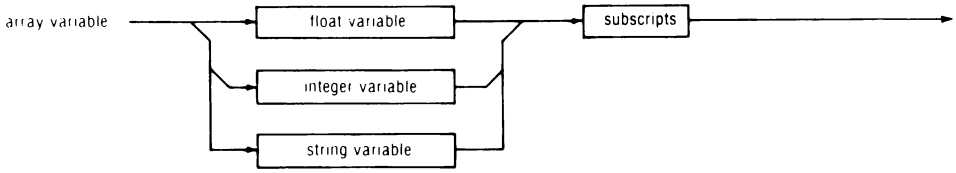
Format

legal variable name (row%, column%)

Syntax Diagram



Description

An array variable is a collection of variable data under one name.

- Arrays consist of floating-point, integer, or string variables.
- The variable name has either one or two subscripts to identify individual items within the array.
- Compiled and Extended BASIC allow three-dimensional arrays. Array variable names may have from one to three subscripts.
- Subscripts are enclosed in parentheses.
- When two subscripts are used, they are separated by a comma.
- It is helpful to view two-dimensional arrays as a matrix. The first subscript is the row number, and the second subscript is the column number. For example, FT%(3,18) identifies the integer in row 3, column 18 of the array FT%(m,n).
- A subrange (portion) of an array can be designated by specifying a first and last subscript separated by two periods. For example, FT% (1..3, 15..18) identifies all integers from row 1 to row 3, columns 15 through 18 of the array FT%.
- Subranges may only be used with certain I/O statements (PRINT, INPUT, RBYTE, WBYTE, RBIN, WBIN, READ). Subranges specify the array elements to be read or written.

ARRAY Variable

- Array variables are distinct from simple variables. A and A(0) are two different variables.

- Only one array variable can be associated with an identifier. A%(n) and A%(m,n) are not simultaneously allowed.

- Memory space must be reserved for an array variable before it can be used. See the discussion of the DIM statement.

- Virtual arrays are array variables accessible through a channel to a file-structured storage device. This feature allows a program to take advantage of the much greater storage space available on these mass storage devices. Refer to the section on Data Storage.

- Some examples of array variables are:

```
A%(3)
B1%(2%, 3%)
A$(5)
C(3%)
D(2 + A * B, C)
D( D(2) )
```

```
A$(3..7)          Strings 3 through 7 of the string array A$.

FT%(2..4, 5..15)  Rows 2 through 4 in columns 5 through 15 of
                  the integer array FT%.
```

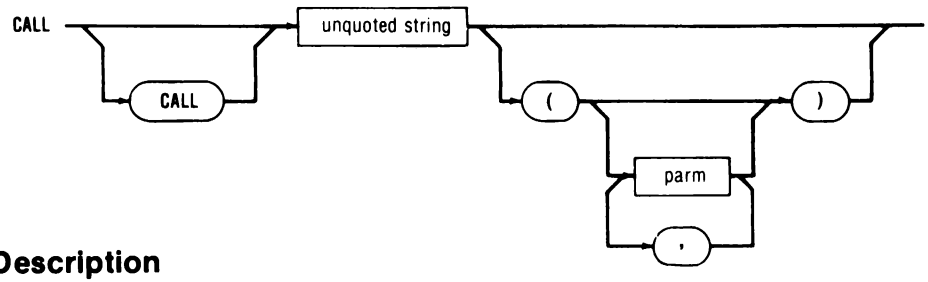
- In the last example above, the second subscript is incremented or decremented before the first. For example, the statement PRINT FT%(2..4,5..15) will display the range FT%(2, 5 through 15) before displaying FT%(3, 5 through 15).

CALL 10 Statement

Usage

CALL {unquoted string} (optional parameter list)

Syntax Diagram



Description

The CALL statement executes a subroutine file loaded by the LINK statement.

- The CALL statement can be used in either Immediate or Run mode.
- As shown in the syntax diagram, the CALL statement verb is not required. This is called "an implied CALL". If CALL is not used, then the leading characters of the unquoted string must not conflict with a BASIC verb.
- The unquoted string contains the subroutine name. The subroutine must be present in memory when the CALL statement is executed. If the subroutine is not present, error 705 (Call to undefined subroutine) is displayed.
- Zero (0) to 10 parameters can be passed to a subroutine as part of a CALL statement.

  1. Parameters can be:

     a. Variables

     b. Constants

     c. Expressions

     d. Arrays (but not virtual arrays)

     e. Individual array elements

2. The parameter list must be enclosed in parentheses.

3. Individual parameters must be separated by commas.

4. The required parameter format is described in the Subroutine section of the BASIC manual.

- The number of parameters that can be exchanged with the calling routine is not limited to 10 when the CALL statement is used with Compiled or Extended BASIC.

- Compiled BASIC subroutine names may be of any length, but are limited to 6 significant characters. All subroutine names are converted to upper case in the output file.

In Extended BASIC Only

- Global variable and subroutine names may be of any length, but are limited to 8 significant characters. All names are converted to upper case in the .OBX file.

- BASIC CALL statements that refer to machine code subroutines should use names that are not more than 6 characters long. Names longer than 6 characters will not be matched by the Extended Library Linker program.

- To simplify programming, use the following rules when Assembly or FORTRAN routines are called:

  1. Because Extended BASIC converts subroutine names to upper case, Assembly programs should use upper case labels for entry points that may be called from BASIC. This is not a problem in FORTRAN, where the compiler always generates upper case labels for subroutines.

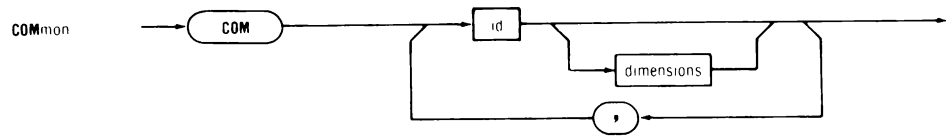
  2. The subroutine name used in a CALL statement to a FORTRAN or Assembly routine should be less than or equal to 6 characters long.

COM 16 Statement

Usage

COM {id list}

Syntax Diagram



Description

COM reserves variables and arrays in a COMmon area for reference by chained programs. COM arguments are valid BASIC array, integer, and floating-point variable names in a comma separated (",") list. Array variables must include their size declaration.

- Only floating-point and integer variables may be stored in the common area.
- String variables may not be stored in the common area.
- String variables may be stored in virtual arrays for access by chained programs. This technique is discussed in Section 7 of the BASIC manual.
- All programs accessing a common area must use COM statements that are identical in order, type, and array sizes; the actual variable names, however, may be different.

Example

For example, assume that a chained program requires the use of three floating point simple variables, an integer simple variable, a floating-point array, and an integer array defined in a previous program. The first program could use a COM statement such as:

```
10      ! Program A
20      COM  A, B, C, F%, D(24%), T%(100%)
        .
        .
        .
1050    RUN "B"
1060    END                              ! End of program A
```

The second program could then use:

```
10      ! Program B
20      COM L1, L2, L3, Q%, K(24%), P%(100%)
        .
        .
        .
```

Note that while the names of the variables stored in the common area have changed between programs, the order and type of the variables are exactly the same.

Differences in Extended and Compiled BASIC

In addition to sharing variables between chained programs, Extended and Compiled BASIC programs also use the COM statement to make certain variables accessible to both calling routines and true subroutines.

- The COM statement must be used first in the Main program segment to assign variables as Common ones. After that, any subroutine may have access to these variables referring to them as Common.

- All COM statements (whether in a main program, a subroutine, or a chained program) must allocate the same amount of space (the same number of variables of the same type).

- No strings or entire arrays may be exchanged via the COM statement.

COM Statement

Extended and Compiled BASIC Examples

In the Main program, this statement

```
COM A, B%(40)
```

assigns the two items A, and the array B%, with 41 elements, as Common to all program sections. Later, a subroutine may use all or some of these variables with another COM statement. In this example, this statement

```
COM Q, G%(40)
```

in a subroutine specifies that Q and the array G%, with 41 elements, from the Main program segment may also be used in this subroutine.

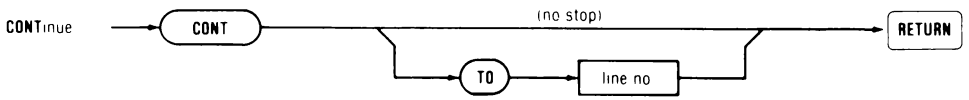
Note that the actual names used in the subroutine COM statement are not significant. In the example above, the subroutine's variable Q is the same as A in the Main program.

CONT TO 18
Immediate Mode Command

Usage

CONT
CONT TO {linenumber}

Syntax Diagrams



Description

The Immediate Mode CONT TO line number command causes program execution to continue from a breakpoint stop caused by STOP, STOP ON, CONT TO, or ⟨CTRL⟩/C.

- The CONT TO line number command is available only in Immediate Mode.
- CONT TO line number must first be enabled by a breakpoint stop in a running program, caused by STOP, STOP ON, CONT TO, or ⟨CTRL⟩/C.
- Any subsequent action other than entering the CONT TO line number command disables the command. The program must then be rerun to the breakpoint.
- Program execution continues at the statement following the last statement executed.
- When TO line number is included, program execution again stops if the specified line number is encountered, and the statement is not executed.
- CONT TO can be used instead of or in addition to STOP ON and CONT to move quickly through a subroutine or loop that has already been confirmed during Step Mode logic debugging.

CONT TO
Immediate Mode Command

Example

The following example program is in main memory during the interaction that follows.

```
10      FOR I% = 1% TO 2%
20      PRINT I% + I%
30      NEXT I%
40      PRINT "Done!"
50      END
```

With the above program in main memory, the following sequence of commands and RETURN entries would produce the responses shown. Programmer entry is shown to the left, and Controller response is shown to the right.

PROGRAMMER ENTRIES	CONTROLLER RESPONSES
	Ready
STOP ON 10	
	Ready
RUN	
	Stop at line 10
	Ready
STEP	
	Stop at line 10
	Ready
(RETURN)	
	Stop at line 20
	Ready
(RETURN)	
	2
	Stop at line 30
	Ready
(RETURN)	
	Stop at line 20
	Ready
(RETURN)	
	4
	Stop at line 30
	Ready
(RETURN)	
	Stop at line 40
	Ready
(RETURN)	
	Done!
	Stop at line 50
	Ready
STOP ON 10	
	Ready
RUN	
	Stop at line 10
	Ready
CONT TO 40	
	2
	4
	Stop at line 40
	Ready
CONT	
	Done!
	Ready

Remarks

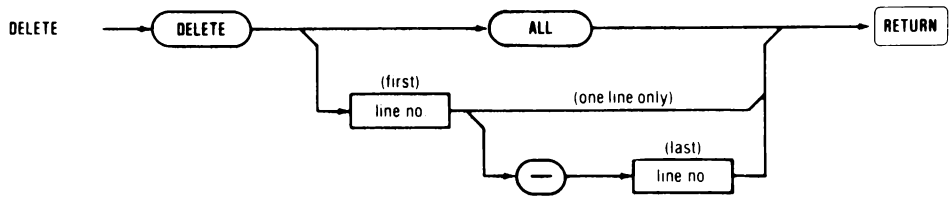
This command is not used in Compiled or Extended BASIC. If you attempt to use this command, the BASIC Compiler will report a syntax error.

Immediate Mode Command

Usage

DELETE {ALL}
DELETE {linenumber}
DELETE {from linenumber - to linenumber}

Syntax Diagram



Description

The DELETE command deletes part or all of a program from memory.

- The entire program is deleted when ALL is specified.
- DELETE ALL also deletes Common Variables (see the COM statement in the Program Chaining section of the BASIC manual).
- One line is deleted if a single line number is specified. The command is ignored if the line does not exist.
- One line may also be deleted by typing the line number only, followed by pressing RETURN.
- The portion of the program between and including specified lines is deleted if two line numbers are specified.

DELETE
Immediate Mode Command

Examples

The following examples illustrate common uses of the DELETE command.

COMMAND	RESULTS
DELETE	No action
DELETE ALL	Deletes entire program and the COM variables
DELETE 100	Deletes only line 100
DELETE 200-300	Deletes lines 200 through 300
400 RETURN	Deletes line 400

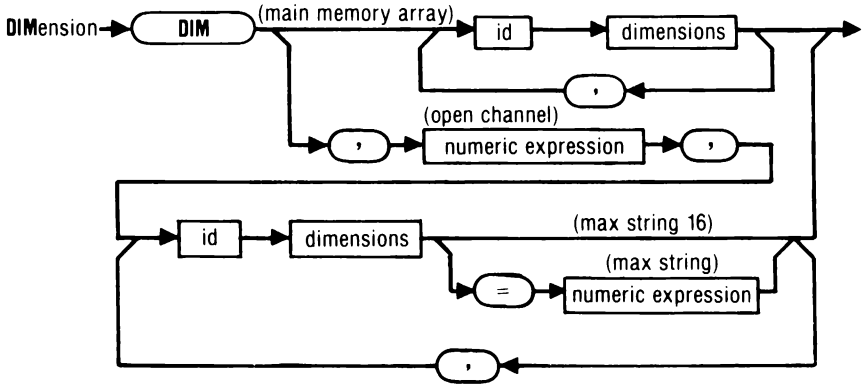
Remarks

This command is not used in Compiled or Extended BASIC. If you attempt to use this command, the BASIC Compiler will report a syntax error.

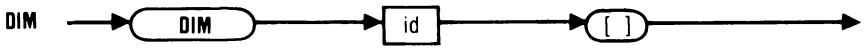
Usage

DIM [id(dimensions)]
DIM #[open channel],[id(dimensions)]
DIM #[open channel],[id(dimensions) = length]

IBASIC Syntax Diagram



Additional CBASIC and XBASIC Syntax



Description

The DIM statement reserves memory or file space for arrays in main memory or on a file-structured device. The DIM statement has the following characteristics when used for a main-memory array.

- An array is a set of variables.
- Each variable is an "element" of the array.
- The maximum array index for each dimension may be specified as one less than the required number of elements, as element counting begins with zero.
- One or two dimensions may be specified.
- Compiled and Extended BASIC allow three-dimensional arrays. See below.
- A two-dimensional array may be considered as a matrix with the first dimension representing rows and the second dimension representing columns.
- A single DIM statement may dimension one or more arrays, separated by commas.

DIM
Statement

The DIM (DIMension) statement may also be used to assign a previously opened channel to a virtual array and informs BASIC about data organization within the file. Virtual arrays are stored on a file-structured device and are treated as random-access files. The DIM statement has the following characteristics when used to describe a virtual array.

- The # character and numeric expression following DIM specify an open channel from 1 to 16.
- String array declarations may specify the maximum length, in characters, of each element string.
- This length specification follows the array identifier and array size specification. For example, DIM #4, Q$(63%, 63%) = 8% declares Q$ to be a virtual array, through channel 4, containing 64 X 64 string elements of 8 characters each.
- String element lengths in virtual arrays must be a power of 2 between 2 and 512 (2, 4, 8, 16, 32, 64, 128, 256, or 512).
- 16 characters per string is assumed when no length is specified.
- The virtual array DIM statement does not initialize string or numeric variables to nulls or zeros as does the ordinary DIM statement. For this reason, a value must be assigned to virtual array elements before they can be used as source variables. After being dimensioned they contain whatever bit patterns are in their respective disk storage areas.

Examples

Main Memory Arrays

The following examples illustrate some common uses of the DIM statement, with comments on the results of each statement.

	MEANING
DIM A(5)	Dimensions a six-element one-dimensional floating point array with the following elements:

```
A(0)
A(1)
A(2)
A(3)
A(4)
A(5)
```

DIM A%(2, 3) Dimensions a twelve-element two-dimensional integer array with 3 rows (0-2) and 4 columns (0-3):

```
A%(0, 0)    A%(0, 1)    A%(0, 2)    A%(0, 3)
A%(1, 0)    A%(1, 1)    A%(1, 2)    A%(1, 3)
A%(2, 0)    A%(2, 1)    A%(2, 2)    A%(2, 3)
```

DIM OP$(5) Dimensions a six-element string array that could be used to store operator messages.

DIM A(5), A%(2,3), OP$(5)

This statement accomplishes the tasks of the three previous examples in one statement.

Virtual Arrays

In the following example, line 10 specifies that the virtual array file "RESULT.VRT" will be 20 blocks long and accessible on channel 1. Line 20 assigns four virtual arrays to the open channel 1.

```
10   OPEN "RESULT.VRT" AS NEW DIM FILE 1 SIZE 20
20   DIM #1, A$(63%) = 128%, C$(31%), B(40%), A%(500%)
```

Note that the data will fit in 20 blocks, but not in 19:

ARRAY	# OF ELEMENTS	SIZE	# OF CHARACTERS
A$	64	128	8192
C$	32	16	512
B	41	8	328
A%	501	2	1002

TOTAL: 10034 bytes (or characters) dupl

19 blocks = (19 * 512) = 9728 characters
20 blocks = (20 * 512) = 10240 characters

DIM
Statement

Using Arrays in Compiled and Extended BASIC

In Compiled and Extended BASIC, the DIM statement is modified to allow variable arrays to three dimensions. The modified DIM statement also allows the exchange of arrays by reference between calling routines and true subroutines.

Three-Dimensional Arrays

The DIM Statement may be used to dimension arrays up to three dimensions.

For example, this statement:

```
DIM A(5,4,8)
```

defines a three-dimensional array of six elements (0 through 5), with each element containing five sub-elements (0 through 4), and each sub-element containing nine sub-sub-elements (0 through 8). Refer to the supplementary syntax diagram for "dimensions" in Appendix A of the Compiled BASIC or Extended BASIC manuals. The diagram above should be used in combination with the syntax diagram in Appendix A.

Passing Arrays to Subroutines

Arrays may be passed by reference to subroutines. There are two ways to do this. The first technique is known as conformal dimensioning. This is the preferred method.

The following program segment is an example of conformal dimensioning:

```
100 SUB ABC(KX(), I%)
110 DIM KX()
120 ...
```

Line 100 defines a true subroutine named ABC with two parameters exchanged between it and its calling routine (see SUB Statement). The DIM statement in line 110 specifies that the variable array K%() will have the same number of dimensions (subscripts) and the same dimension limits as it does in the calling routine.

- The variable array may be a normal array in Main Memory, in Common memory, or a virtual array.

- The number of dimensions (subscripts) used in the subroutine must match the dimensions in the array. For example, if K%() is a two-dimensional array, but the subroutine tries use it as a one-dimensional array, an error will result.

- The dimension limits of the array will be the same in the subroutine as in the calling routine. In the above example, if the original dimension on K%() was

  DIM K%(10)

  in the calling routine, an attempt to use, for example, array element K%(11), will cause a Subscript Out of Range error.

- The data type of the original array will be used in the subroutine. In the example, K%() is an array of integer variables. An attempt to use this array for another data type (floating-point for instance) will result in an error.

- Conformal dimensioning may be used across more than one level of subroutines. For example, a Main program segment calls Subroutine A, which in turn calls Subroutine B. A DIM statement Subroutine B may be used to access an array that was originally dimensioned in the Main program segment. The array does not have to be used in Subroutine A in order to be used in Subroutine B.

- An array may be originally dimensioned in a subroutine, then dimensioned conformally in subsequent subroutines that are called from it. The original DIM statement does not have to be in the Main program segment.

- This method will NOT work for subroutines that will be called by FORTRAN or Assembly programs. Use the second method, described below, for these applications.

The second method is to dimension a variable array in a subroutine is to redefine the array dimensions in the subroutine. For example, the following statements

```
200 SUB ABC (KZ(), IZ)
210 DIM KZ (100Z)
220 ...
```

Line 200 defines a true subroutine named ABC with two parameters exchanged between it and its calling routine (see SUB Statement). The DIM statement in line 210 dimensions array K%() to a 101 element single-dimension array.

- The new dimensions for the array only apply within this subroutine.
- Virtual arrays may not be redimensioned with this method. An Illegal Parameter DIM error (error 908) will occur if an attempt is made to redimension a virtual array.
- This method MUST be used if the subroutine will be called by a FORTRAN or Assembly program.

If a subroutine uses a virtual array element, but conformal dimensioning is not desired, the following method may be used:

```
300 SUB ABC (KX(), IX, JX)
310 DIM #IX, KX(JX)
320 ...
```

Line 300 defines a true subroutine named ABC, which exchanges three items with its calling routine (see SUB Statement). The DIM statement in line 310 dimensions the array K%() and assigns a channel number by means of the values passed from the calling routine.

- An Illegal Parameter DIM error (error 908) will occur if the parameter is not a virtual array or if the array is attached to a channel other than the one specified in the DIM statement.

EDIT 30

Immediate Mode Command

Usage

EDIT [linenumber]

Description

The editor provided as a part of the Fluke BASIC Interpreter program is an easy-to-use character-oriented editor. The Edit Mode allows the user to create, delete, or modify the characters that make up program lines in main memory. Program lines are stored in main memory for subsequent use by other modes. The editing keys in the upper right corner of the keyboard plus the ⟨CTRL⟩/U, BACK SPACE, RETURN, and LINE FEED keys control the cursor and delete text. The remaining keys are used for text entry. This section describes the edit keys and their use along with other editor features.

- Edit Mode is entered from Immediate Mode by typing:

  EDIT ⟨RETURN⟩
  or
  EDIT line number ⟨RETURN⟩

- Editing begins with the lowest numbered line of the program in memory unless another line is specified.

- No line number specification is used when beginning the edit of a new program when no other program is in memory.

- Program entry procedure is the same as in Immediate Mode.

- Immediate Mode commands and program statements cannot be executed while in Edit Mode.

- Exit from Edit Mode to Immediate mode by entering ⟨CTRL⟩/C.

- The special edit keys on the programmer keyboard are enabled.

- Up to 15 lines of the program in memory are displayed, beginning at the first line or at the line number given with the command.

- Edit Mode enables the user to scroll the cursor forward or backward in a program as well as right or left on program lines.

EDIT
Immediate Mode Command

- Edit Mode enables the user to delete characters, portions of lines, or entire lines.

- Edit Mode also enables the user to duplicate entire program lines. The following examples illustrate the two different uses of the EDIT command.

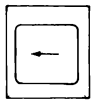
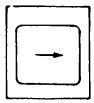
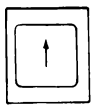
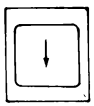
COMMAND	RESULT
EDIT	Select Edit Mode and display up to 15 lines of the program in memory, beginning with the first line. If no program exists, the display is cleared and the cursor is positioned to the upper left corner of the display.
EDIT 1000	Select Edit Mode and display up to 15 lines of the program in memory, beginning with the first line greater than 999. If no program exists or the last line number is less than 1000, the display is cleared and the last line of the program is displayed on the top of the screen.

Edit Mode Keys

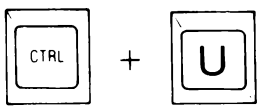
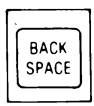
Some of the keys on the programmer keyboard have special functions that are enabled or modified in Edit Mode. Any key, if held down, performs its function repeatedly. The figure below describes the special functions of the Edit Mode Keys.

NOTE

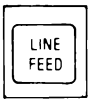
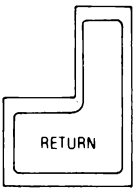
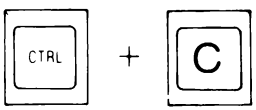
Any edit command that will move the cursor from the current line or ⟨CTRL⟩/C is not accepted if the line does not pass a check for correct syntax. A blinking error message (e.g.: "Mismatched Quotes") will be displayed until the line is corrected.

KEY	ACTION
	Move one position left. Ignored if already at the left margin.
	Move one position right. Ignored if already at the right end of the line.
	Move one position up. If the line above is shorter than the current column position, move left to the end of that line. Scroll down one line if the cursor is on the top line of the display, and another line is available. This action will not be done if the line does not pass a syntax check.
	Move one position down. If the line below is shorter than the current column position, move left to the end of that line. Scroll up one line if the cursor is on the bottom line of the display, and another line is available. This action will not be done if the line does not pass a syntax check.

EDIT
Immediate Mode Command

KEY	ACTION
	Delete from the cursor position to the end of the line. If the cursor is at the left margin, delete the entire line and move the rest of the program up one line to fill the deletion.
	Delete the character at the cursor position and move the remaining characters left one position to fill the deletion. When the key is held down for repeat, the portion of the line to the right of the cursor will progressively disappear.
	Delete the character to the left of the cursor position and move the remaining characters left one position to fill the deletion. When the key is held down for repeat, the portion of the line to the left of the cursor will progressively disappear as the portion to the right moves to the left margin. This key function is also available in Immediate Mode.
	Delete the current line.
	Move to the left margin.

EDIT
Immediate Mode Command

KEY	ACTION
	Move to the right end of the line.
	When the cursor is at the right end of the line, open a new empty line below, and move to its left margin. When the cursor is not at the right end of the line, break the line into two lines. The cursor position identifies the first character of the new (second) line. This action will not be done if the portion of the line that was to the left of the cursor does not pass a syntax check.
Character Keys	Insert characters at the current cursor position. Each character entry moves the cursor right one position along with any text to the right of the cursor. Entries that would result in a line length greater than 79 characters are not accepted, and produce a beep sound.
	Return to Immediate Mode.

EDIT
Immediate Mode Command

Additional Editor Features

It is not necessary to insert a line in correct sequence. Regardless of the order in which program lines are entered, the editor will store them in memory in the correct line number sequence. When the cursor is instructed to move from a line, the editor checks for some syntax errors, such as omitting a quote, parenthesis, or line number. If a line does not pass the check, an error message is displayed and the cursor is not allowed to leave the line until the error is corrected.

If a program line is renumbered by deleting all or part of its line number and then entering a new line number, a duplicate line will result. One line will have the original line number, the other line will have the new line number. This may be seen by scrolling the modified line on and off the display, in EDIT mode.

If ⟨CTRL⟩/C is entered when the current line will not pass the syntax check, the blinking error message is displayed in Immediate Mode and the line is not stored in memory.

There are many errors the editor will not detect, such as forgetting to dimension an array or specifying GOTO with a nonexistent line number. Such errors will be detected only when the program is run.

The cursor will not scroll above the lowest line number nor below the highest line number in the program. If the cursor is in the middle of the program and a new last line is entered at that position, the cursor will not scroll down past that line. To correct this condition:

- The cursor may be scrolled in the opposite direction until the line entered out of sequence disappears from the display. Reverse scroll direction again and the line will then be in proper sequence.

- Type ⟨CTRL⟩/C, and then type EDIT, followed by the line number that needed editing. Lines will then be displayed in correct sequence, allowing access to all lines.

EDIT
Immediate Mode Command

The editor stores program lines in memory with the line number shown on the display. If any other program line has the same line number as that shown on the display, it is replaced with the contents of the line shown on the display. This feature can be used to duplicate program lines by changing only the line number and moving the cursor off the line. The line with the previous line number is not deleted by this process. The display, however, will show only the most recent line number entered. To see both resulting lines, scroll the entered line off the display and back on.

When a line is scrolled off the display with the same line number as a line previously stored, the original line in memory will be replaced by the one which is scrolled off. In order to prevent this from occurring, assign each line a unique line number.

Remarks

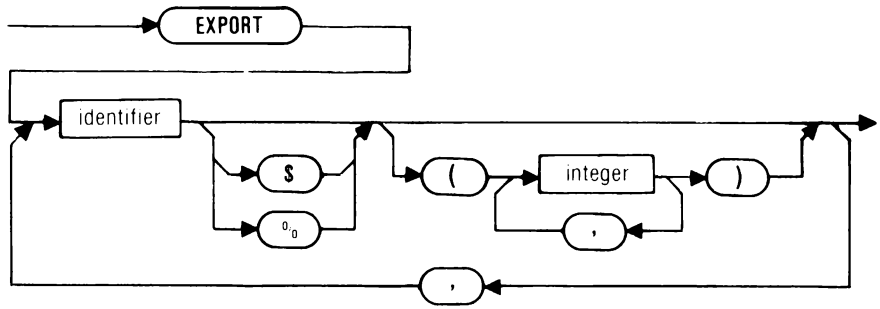
This command is not used in Compiled or Extended BASIC. If you attempt to use this command, the BASIC Compiler will report a syntax error.

EXPORT 35A Statement

Usage

EXPORT {variable list}

Syntax Diagram



Description

The EXPORT statement is used to declare a set of variables and arrays as global variables. EXPORT assigns memory to the variables and arrays in the list of variables and publishes the variable names in the output .OBX file as global variable definitions.

- EXPORT statements can be used in BASIC main programs or true subroutines (bracketed by SUB and SUBEND).
- Variables and arrays in the variable list may be integer, floating-point, or string variables.
- Arrays must be DIMensioned so that the Extended Basic Compiler can notify the Extended Linking Loader to reserve storage space for them. Dimension values must be constant integers or integer-valued real numbers.
- A variable may not be named in both an EXPORT statement and a DIM, COM, IMPORT, or another EXPORT statement in the same program module.
- Any number of EXPORT statements may appear in a single program module.

EXPORT
Statement

- A global variable should be declared in an EXPORT statement before it is used in another BASIC statement in the same program module.

- Global variables are always preset to zero (for numbers) or the null string (for strings) when the program begins execution.

Example

The following example illustrates the use of the EXPORT statement:

```
10 sub get_readings(howMany%)
20 export dvmReadings(100)
30 import dvmAddress%, dvmGetData$
40 print @dvmAddress%, dvmGetData$
50 for i% = 1% to howMany%
60 input @dvmAddress%, dvmReadings(i%)
70 next i%
80 subend
```

EXTENDED IF 36A
Statement

Usage

IF [expression] THEN
  program statements
ELSIF [expression] THEN
  program statements
ELSIF [expression] THEN
  program statements
ELSE [expression]
  program statements
ENDIF

Syntax Diagram

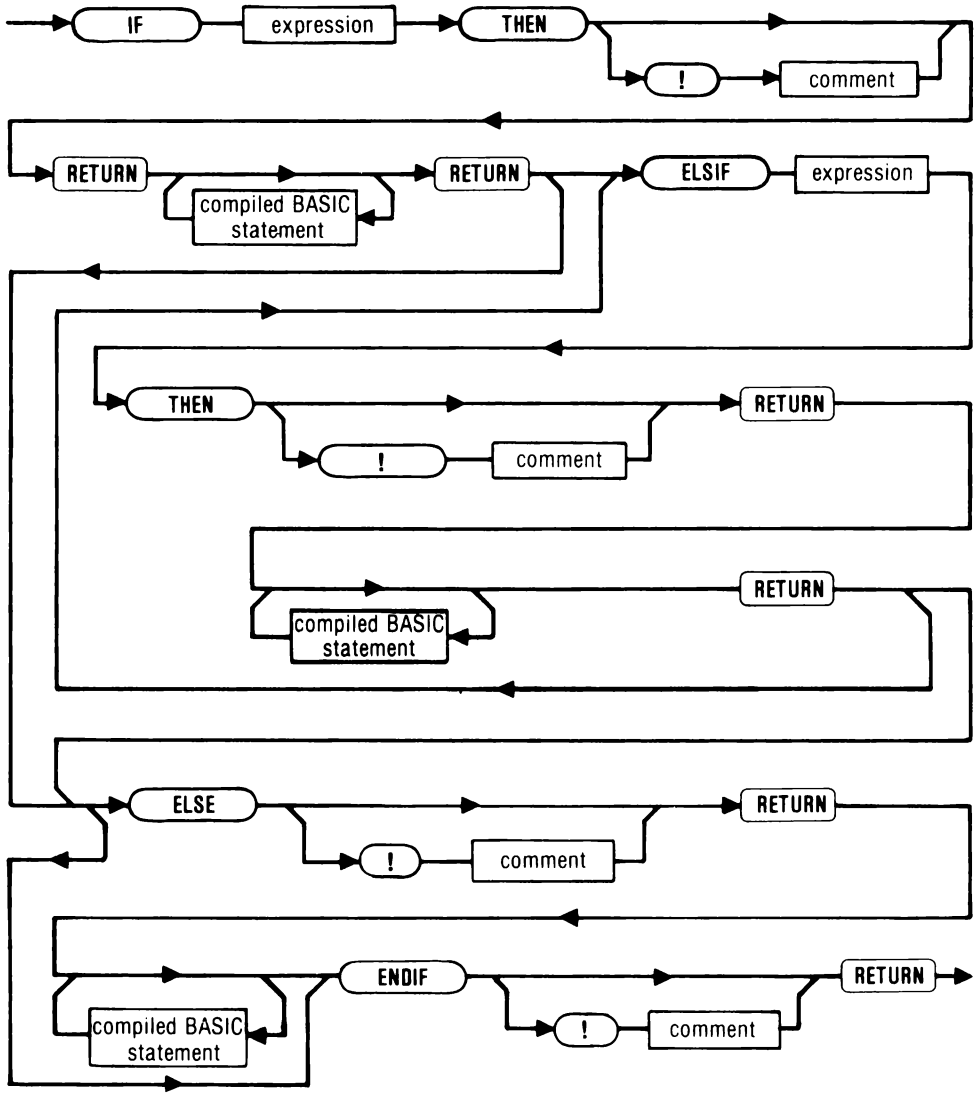
See page 2

Description

The extended IF statement extends the IF-THEN-ELSE statement by allowing the ELSE condition to be the next IF condition and by allowing multiple statement lines between the THEN and ELSE clauses. For example,

```
begin:
      if a = 0 then
           call sub0 ()
      elsif a = 1 then
           call sub1 ()
      elsif a = 2 then
           call sub2 ()
      elsif a = 3 then
           call sub3 ()
      else
           print "selection not possible"
           print cpos(15,10)"Touch Screen to Continue "
           k% = key \ wait 00:01 for key
           goto begin:
      endif
```

EXTENDED IF
Statement



EXTENDED IF
Statement

A extended IF is distinguished by an IF statement which has nothing (except an optional trailing remark and the end of the line) following the THEN keyword. The entire extended IF consists of the following elements:

- An "IF expression THEN" statement which starts the IF. The statements following the THEN are executed if the expression is true (nonzero).

- Zero or more "ELSIF expression THEN" statements which select one of several alternatives if the initial "IF expression THEN" clause was not selected. An ELSIF statement is executed if the expression is true (nonzero). The ELSIF statements are executed in the order in which they appear in the source program.

- An optional "ELSE" statement which is executed if none of the other alternatives was selected. Note that at most one ELSE may be used with any given extended IF.

- A mandatory "ENDIF" statement which terminates the extended IF.

- Extended IF statements must be the only statements in a program line. This restriction eliminates any interaction with "simple IF" statements which are still part of the Compiled and Extended BASIC languages.

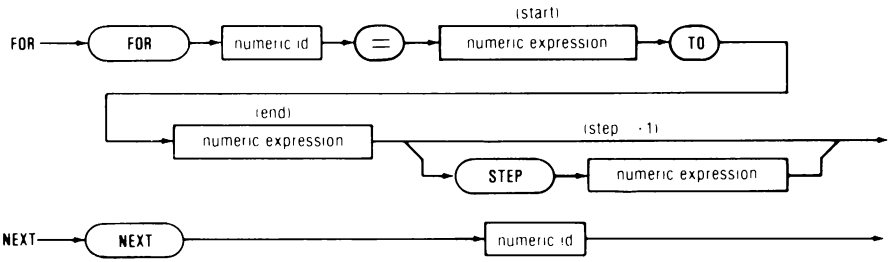
- Extended IF statements may be nested to any desired depth; no confusion (at least where the compiler is concerned) is possible. We recommend that you indent the program text to make it easier to determine which code is under the control of which IF.

FOR and NEXT Statements

Usage

FOR {index} = {begin} TO {end} [STEP {step}]
program steps
more program steps
etc...
NEXT {index}

Syntax Diagram



Description

The FOR statement sets up a loop which repeatedly executes the statements contained between the FOR statement and the NEXT statement.

- There must not be a FOR without a NEXT, or a NEXT without a FOR.
- The FOR statement specifies the number of times the loop is to be repeated by specifying limit points and step increment of the index.
- If no step increment is specified, step +1 is assumed.
- The index must be numeric.
- The index must not be an array element.
- When a FOR statement is initially encountered, the index value is compared to the limit. If the index is past the limit, the FOR-NEXT loop is not executed.
- The NEXT statement compares the index with the limit after incrementing it.

FOR and NEXT Statements

- Using a GOTO statement to branch permanently out of a FOR-NEXT loop is a potential time bomb (see Examples, below.) A better solution is to test for the loop exit condition, then set the value of the loop counter variable to the value of the TO portion of the FOR statement, then branch to the NEXT statement. This applies only to the BASIC interpreter.

- Branching permanently out of a FOR-NEXT loop via a GOTO statement is permitted in Compiled and Extended BASIC.

- When two FOR-NEXT loops are nested, each FOR statement must have a corresponding NEXT statement. Furthermore, the innermost FOR statement must match the innermost NEXT statement, and so on, to the outermost FOR-NEXT statement pair.

- Compiled and Extended BASIC do not allow multiple NEXT statements within a FOR-NEXT loop. Each FOR statement must have one and only one NEXT statement.

Examples

The following examples compare two different ways of constructing program loops, using the FOR - NEXT, and using IF - GOTO. Note the different comparisons at line 100, depending on the sign of the STEP.

FOR - NEXT

```
10 FOR I = 1 TO -5 STEP -2
20! Other statements
30!
100 NEXT I
110 ! Other statements

10 FOR J = 10 TO 100 STEP 20
20! Other statements
30!
100 NEXT J
110 ! Other statements
```

IF - GOTO

```
10 I = 1
15 IF I < (-5) GOTO 110
20! Other statements
30!
100 I=I + (-2) \ GOTO 20
110 ! Other statements

10 J = 10
15 IF J > 100 GOTO 110
20! Other statements
30!
100 J=J + 20 \ GOTO 20
110 ! Other statements
```

The following example loops five times, and then prints the index value. Note that the index has incremented to six.

```
10  FOR I%=1% TO 5%                  ! Increments by 1 through loop
20  PRINT I%                         ! Display value of I%
30  NEXT I%                          ! Repeat loop until F=5%
40  PRINT "Index Value is"; I%
```

The display will show:

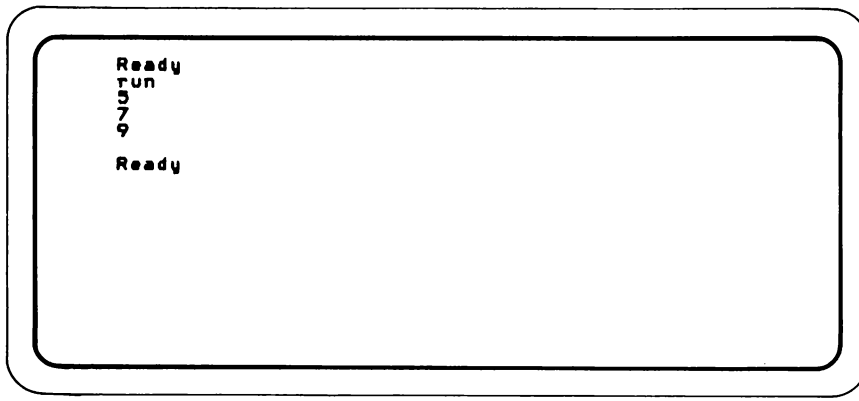
```
Ready
run

 1
 2
 3
 4
 5
Index value is 6

Ready
```

FOR and NEXT Statements

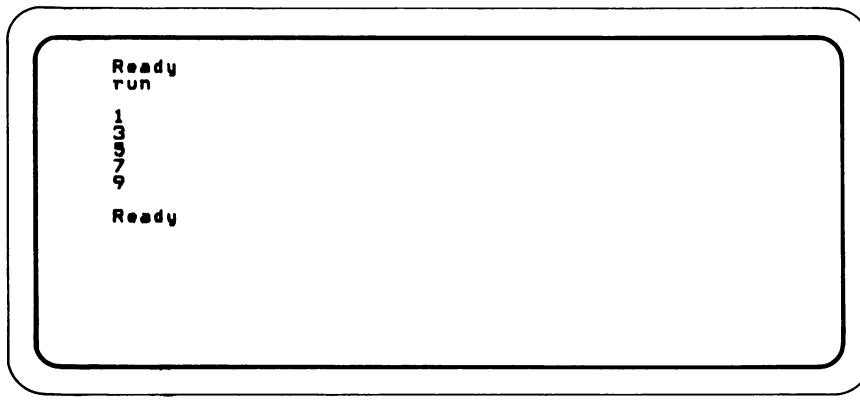
The following example illustrates what happens when the index never equals the final value.

```
10  FOR A% = C% TO D% STEP 2%        ! C% = 5 and D% = 10
20  PRINT A%                         ! Display value of A%
30  NEXT A%                          ! Increments A% by
40  REM                              ! Loops until A% = 10
```

The display will show:



If line 10 used C and D rather than C% and D%, and if C = .6 and D = 10.6, the display would show:



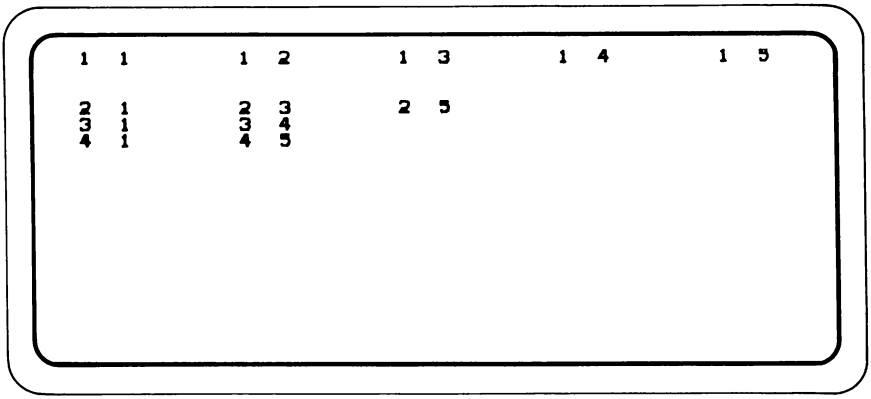
because real values are rounded before assignment to an integer variable.

FOR and NEXT Statements

The following example illustrates the nesting of two FOR - NEXT loops. Note that one blank line occurs between first and second display lines. This happens because the last value displayed in the line above did not have 16 columns available, causing a carriage return-line feed in the display (See PRINT command).

```
10 FOR I% = 1% TO 4%
20       FOR J% = 1% TO 5% STEP I%
30       PRINT I%, J%,                    ! Print values and then tab
40          REM                           ! to next column
50       NEXT J%
60       PRINT                            ! Move to the next line
70 NEXT I%
80 REM INDENTING SHOWN FOR CLARITY
```

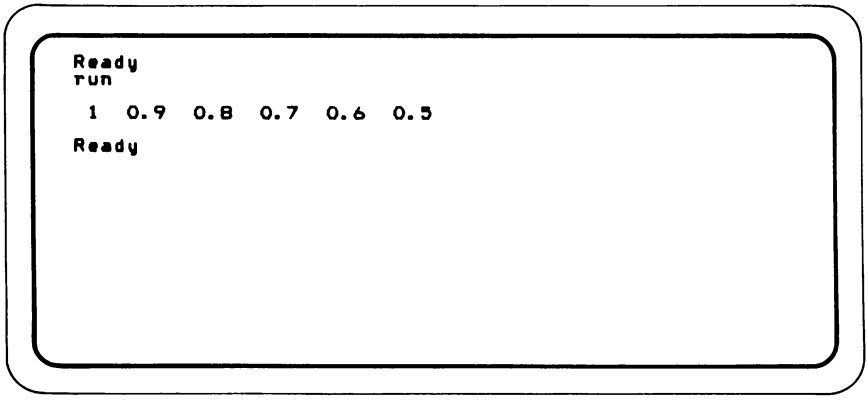
The display will show:



The following example decrements the index by -.1.

```
10  FOR I = 1 TO 0.5 STEP -.1
20  PRINT I;
30  NEXT I
```

The display will show:

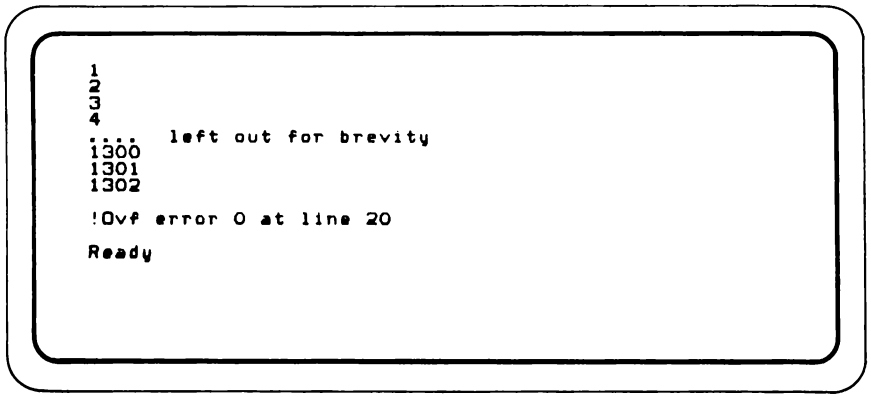


FOR and NEXT
Statement

The following example shows what happens when a GOTO statement branches out of a FOR-NEXT loop and returns to the loop beginning.

```
10   for x = 1 to 100000
20   y = y + 1
30   print y
40   goto 10
50   next x
60   end
```

Which results in the following display:

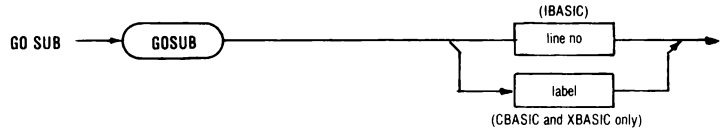


GOSUB 40 Statement

Usage

GOSUB {linenumber}

Syntax Diagram



Description

The GOSUB statement transfers control to the beginning of a subroutine. The RETURN statement terminates the subroutine and returns control back to the statement following the GOSUB.

- BASIC allows nested subroutines.
- Within a subroutine, a GOSUB may call another or the same subroutine.
- A GOTO within a subroutine should not pass control permanently to a program segment outside of the context of the subroutine.
- Any valid BASIC statements may be used in the body of a subroutine. Available memory is the only limit to the length of subroutines.
- In Compiled and Extended BASIC, GOSUB causes program flow to branch to the specified program label.

GOSUB
Statment

Example

In the following example the subroutine displays the operator's options and obtains his or her choices. Control is transferred to the subroutine at line 180. The display subroutine calls another subroutine (line 1300) to obtain identification of the selected choice (line 1080). When the RETURN statement at line 1390 is executed, control is transferred to line 1090. Three subroutine options are illustrated, each with different interpretations for GOSUB and RETURN.

The following points should be noted about this example:

- GOTO 31000 causes termination of the test without returning from the subroutine. This normally does not represent good program structure.
- ELSE 1500 causes control to be transferred to lines 1500 - 1590 which can be considered an extended part of subroutine 1000. The RETURN at line 1590 will cause line 190 to be executed next.
- RETURN at line 1090 will cause immediate termination of the subroutine 1000. Line 190 is executed next.

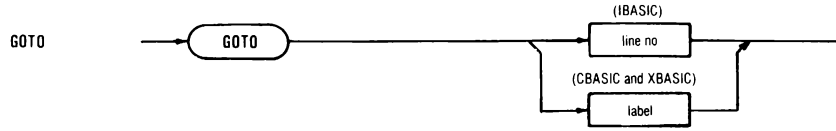
```
100 ! Other statements
180 GOSUB 1000                          ! Get operator choice
190 ! Other statements
999 STOP
1000 REM -- SUBROUTINE -- Display and Execute Operator Options
1010 PRINT "Enter choice by pressing display"
1020 PRINT\PRINT                        ! Skip 2 lines
1030 PRINT "      [Continue test]"      ! KR% = 1 or 2
1040 PRINT\PRINT\PRINT                  ! Skip 3 lines
1050 PRINT "      [Adjust Instrument]"  ! KR% = 3 or 4
1060 PRINT\PRINT\PRINT                  ! Skip 3 lines
1070 PRINT "      [Terminate Test]"     ! KR% = 5 or 6
1080 GOSUB 1300                         ! Get choice, KR%
1090 IF KR%)2% THEN IF KR%)4% GOTO 31000 ELSE 1500 ELSE RETURN
1300 REM -- SUBROUTINE -- Get response from Keyboard
1310 ! On exit KR% will be the row touched (from 1 to 6)
1320 ! KC% will be the column touched (from 1 to 10)
1330 ! Other statements
1390 RETURN
1500 REM -- SUBROUTINE -- Adjust Instruments
1510 ! Other statements
1590 RETURN
31000 REM -- Termination Routine
31010 ! Place instruments in standby state and save test results
31020 ! Other statements
32767 END
```

GOTO Statement

Usage

GOTO {linenumber}

Syntax Diagram



Description

The GOTO statement causes a program to unconditionally branch to the specified line number. In Compiled and Extended BASIC, it also causes an unconditional branch to a program label.

- In Interpreted BASIC, GOTO must not be used to branch out of a FOR-NEXT loop. If GOTO is used in Interpreted BASIC, a user storage overflow (error 0) will eventually result.

- A GOTO statement may be used to begin running a program, starting with any line number in the program. When a GOTO statement is used to run a program, many of the normal actions of a RUN statement are not performed, such as reseting variables and random numbers. This may be useful in program debugging.

- In Compiled and Extended BASIC, GOTO causes the program flow to branch to the specified program label.

Example

The following program example illustrates one use of the GOTO statement. The GOTO statement at line 80 causes line 30 to be executed next:

```
10 REM -- Main Program Routine
20 REM
30 GOSUB 4000                          ! Set-up Sequences
40 GOSUB 5000                          ! Test #1
50 GOSUB 6000                          ! Test #2
60 GOSUB 7000                          ! Test #3
70 GOSUB 8000                          ! Test #4
80 GOTO 30                             ! Start Again
```

GOTO
Statement

Compiled and Extended BASIC Example

The GOTO statement causes program flow to transfer to the specified program label depending upon the value of the string variable name$.

```
foo:

    if name$ = "FLUKE" then
       goto success
    else
       goto failure
    endif
 success:
 ...
 ...
 ...
 failure:
 ...
 ...
 ...
```

Remarks

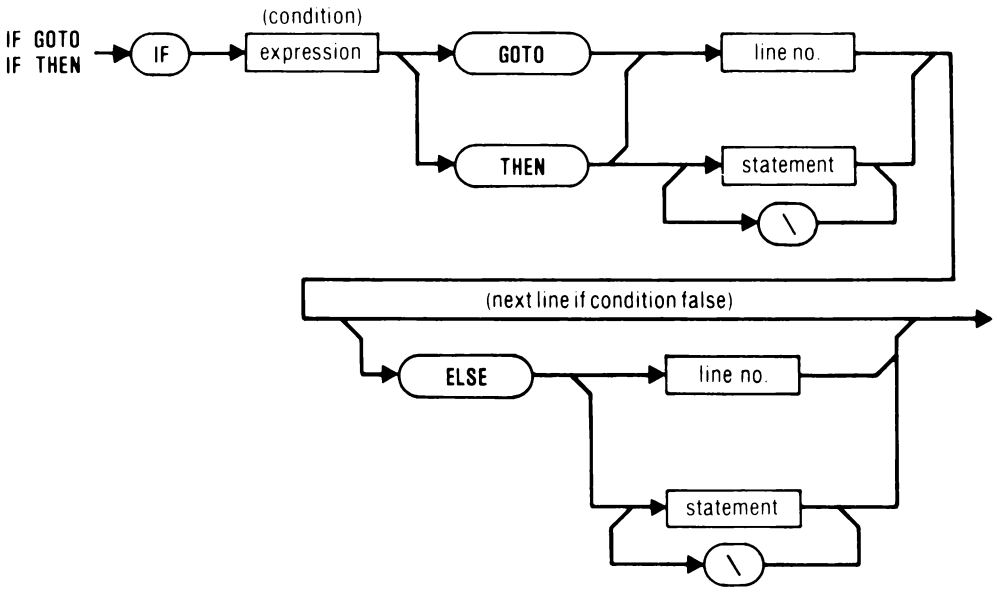
Compare the GOTO statement with the GOSUB statement.

IF-GOTO 42
IF-THEN
IF-THEN-ELSE
Statement

Usage

IF {condition} GOTO {linenumber}
IF {condition} goto statement label !CBASIC only
IF {condition} THEN {BASIC statement}
IF {condition} THEN {BASIC statement} ELSE {BASIC statement}

Syntax Diagram



Description

The IF statement evaluates a condition represented by an expression. The resulting course of action depends on the result of that evaluation and the structure of the statement.

NOTE

Relational expressions such as A=B, A%<3%, A$>=B$ evaluate as -1 if true and 0 if false.

- The conditional expression must result in a value representable as an integer.

IF-THEN-ELSE
Statement

- A non-zero result is TRUE. A zero result is FALSE.
- Control passes to the line number following GOTO, or to the statement or line number following THEN, if the result is TRUE.
- Control passes to the next program line if the result is FALSE and ELSE is not specified.
- Control passes to the statement or line number following ELSE if the result is FALSE and ELSE is specified.
- A line number must follow GOTO, if it is used.
- A line number or a valid BASIC statement must follow THEN, if it is used.
- In Extended and Compiled BASIC, a program statement label may be used in lieu of the line number in any of the forms of the IF statement. The GOTO keyword is required to distinguish a branch to a statement label from a branch to a named subroutine.
- Multiple statements, separated by the "\" character, may be used after THEN, and after ELSE. Each statement will be done in sequence only if control is passed to that portion of the IF statement as defined above.

Examples

The following examples illustrate the results of various uses of the IF statement:

STATEMENT	RESULTS
IF A THEN 100	If A is non-zero, go to line 100. If A is zero go to the next line.
IF A>B THEN A=B	If A is greater than B, set A equal to B. Then go to the next line.
IF NOT A% GOTO 500	If A% equals -1 (logical true), ignore this statement and go to the next line. Otherwise, go to line 500.

IF-THEN-ELSE Statement

IF A%+B% GOTO 500	If A%+B% is non-zero (A% not equal to -B%), go to line 500. Otherwise go to the next line.
IF A%+B% THEN A=B*5B=10	If A%+B% is non-zero (A% not equal to -B%), assign B*5 to A, and assign 10 to B. Then go to the next line.
IF A% OR B% GOTO 500	If either A% or B% is non-zero, go to line 500. Otherwise go to the next line.
IF A+B=C THEN PRINT C	If A+B equals C, display value of C. Then go to the next line.
IF (1<A) AND (A<6) THEN 450	If the value of A lies within the open interval (1,6) transfer control to line 450.

The following program example shows some common uses of the IF statement. The relational expression in line 140 uses the AND operator to ensure that both conditions are true before transferring control to line 5000.

```
100 REM -- Determine Test to Run
110 PRINT "ENTER UNIT SERIAL NUMBER";    !Print prompt
120 INPUT SN%
130 IF SN% ) 12563% THEN 110          !Equivalent to IF...GOTO 110
140 IF (11000% (= SN%) AND (SN% ( 11500%) GOTO 5000
150 REM -- Run Standard Test
160 ! Other statements
5000 REM -- Run Test for 'Special'
5010 ! Other statements
```

IF-THEN-ELSE
Statement

The following example shows how the IF statement acts on various values. Examining the results will aid in understanding the IF statement. Note that line 100 transfers control back to line 30 except when A% is 0.

```
10 REM -- Illustration of IF with integers
20 PRINT "INTEGER VALUE","ONE'S COMPL.","ODD","DIVISIBLE BY FOUR"
30 INPUT A%
40 PRINT A%, NOT A%,                          !Print values and tab
50 IF A% AND 1% THEN PRINT 'YES;              !Is A% odd?
60 PRINT,                                     !Go to next tab on display
70 IF A% AND 3% THEN PRINT; \ GOTO 90         !Is A% divisible
80 PRINT 'YES';                               !by four?
90 PRINT                                      !Line Feed
100 IF A% THEN 30
110 END
```

Results:

```
INTEGER VALUE    ONES COMPL.      ODD              DIVISIBLE BY FOUR
 1               -2               YES
 4               -5                                YES
 1568            -1569                             YES
 9               -10              YES
-568              567                              YES
-1                0               YES
-4                3                                YES
 0               -1                                YES
```

The following example includes the ELSE option. When A is less than 5, B will be set to FNS(A) and the statements following line 200 will be executed until the STOP statement on line 290 is encountered. When A is greater than or equal to 5, B will be set to FNT(A) and the statements following Line 300 will be executed.

```
10 ! Other statements
100 IF A < 5 THEN 200 ELSE 300
200 B = FNS(A)                          ! Use 'S' function
210 ! Other statements
290 STOP
300 B = FNT(A)                          ! Use 'T' function
310 ! Other statements
```

The following example shows the ELSE option followed directly by statements. When A is less than B, J is set to B raised to the power A, and T is set to 0. When A is greater than or equal to B, J is set to the value of A raised to the power B, and T is set to 1.

$$\text{IF A} < \text{B THEN J} = \text{B}\hat{}\text{A}\backslash\text{T} = \emptyset \text{ ELSE J} = \text{AB}\backslash\text{T} = 1$$

The following example illustrates nested IF-THEN-ELSE statements. The ELSE is always associated with the closest IF and THEN statement to the left that is not already associated with another ELSE, as indicated by the diagram below the statement. (Note: This statement computes M = minimum of A, B, or C.

```
IF A<B THEN IF A<C THEN M=A ELSE M=C ELSE IF B<C
THEN M=B ELSE M=C
```

The logic is as follows:

If A is less than B, and then if A is less than C, A is the minimum (M).

If A is less than B, and then if A is greater than or equal to C, C is the minimum (M).

If A is greater than or equal to B, and then if B is less than C, B is the minimum (M).

If A is greater than or equal to B, and then if B is greater than or equal to C, C is the minimum (M).

Extended and Compiled BASIC Examples

The following examples show the use of program statement labels in Extended and Compiled BASIC.

STATEMENT	RESULTS
IF A THEN GOTO DMM	Branch to statement label DMM:
IF X% = 5% THEN &	Branch to label GET DATA:
GOTO GET DATA &	if x% = 5%, otherwise goto
ELSE GOTO BEGIN	label BEGIN:

Remarks

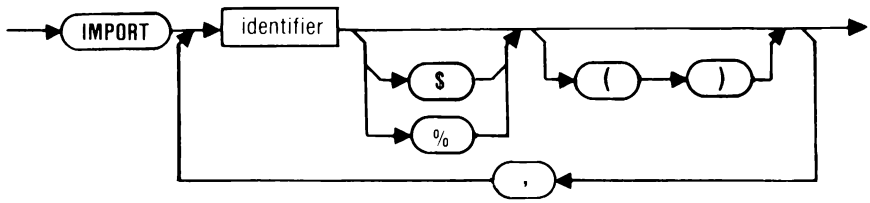
Refer also to the complex IF statement, Reference #36a.

IMPORT 42A Statement

Usage

IMPORT {variable list}

Syntax Diagram



Description

The IMPORT statement is used to declare a set of variables, which have been defined in a BASIC main program or a true subroutine, as global variables. IMPORT publishes the variable names in the output .QBX file as global variable references.

- Variables and arrays in the variable list may be integer, floating-point, or string variables.
- An array declared in an IMPORT statement should use only an empty pair of parentheses "()" in the IMPORT statement to indicate that the variable is an array. No array subscripts may be used in the IMPORT statement itself.
- A variable may not be named in both an IMPORT statement and a DIM, COM, EXPORT, or another IMPORT statement in the same program module.
- Any number of IMPORT statements may appear in a single program module.
- A global variable should be declared in an IMPORT statement before using it in another BASIC statement in the same program module.

Example

The following program illustrates the use of the IMPORT statement:

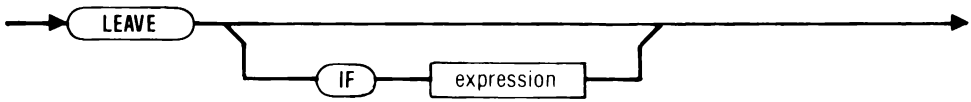
```
10 sub printStrings(start%, finish%, channel%)
20 import array$()
30 for i% = start% to finish%
40 print #channel%, array$(i%)
50 next i%
60 subend
```

LEAVE 59A Statement

Usage

LEAVE
LEAVE IF {expression}

Syntax Diagram



Description

The LEAVE statement is used to leave a structured loop (WHILE, REPEAT, LOOP, or FOR.) The LEAVE statement has two forms:

LEAVE

LEAVE IF {expression}

- The first form (LEAVE) simply jumps out of the innermost loop unconditionally.
- An expression is anything that can be evaluated to true or not true.
- The second form (LEAVE IF) leaves the loop only if the {expression} has a true (nonzero) value.
- The LEAVE statement may be subordinate to an IF or other statement; it simply causes an exit from the innermost loop in which it occurs.

Examples

The following program fragment illustrates the two constructions of the LEAVE statement.

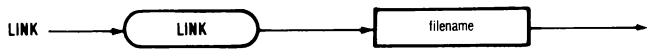
```
LOOP
  PRINT "Enter a value";
  INPUT a%
  IF (a% < 0%) THEN
          PRINT "Negative. Leaving the loop"
          LEAVE
  ELSIF (a% <=10%) THEN
          PRINT "Between 0 and 10"
          LEAVE IF a% = 5%          ! leave for 5
  ELSE
          PRINT "Greater than 10."
  ENDIF
ENDLOOP
```

LINK Statement

Usage

LINK {filename}

Syntax Diagram



Description

The LINK statement loads an object file which contains one or more Assembly Language or FORTRAN subroutine(s) into the Instrument Controller memory.

- The object file is named in the filename shown in the LINK syntax diagram. The filename must contain the file name and may also contain a device name and a file extension. The filename must be enclosed in quote characters (").

  1. If a device name is not specified, the System Device (SY0:) is searched for the file.

  2. If a file extension is not specified, the Instrument Controller uses the default extension .OBJ.

- If the specified file is not found, error code 305 (file not found) is displayed.

- If the specified file is found, the subroutine(s) in the file is(are) loaded into the Instrument Controller memory.

- Multiple LINK commands can be used to load different object files.

- LINK can be used in both Immediate and Run modes.

- LINK commands must follow all COM statements in the program because COM storage must be defined before loading any subroutine(s).

Example

```
14225 LINK "tests.rep"          ! Load the subroutines in the
14226                           ! "tests.rep" object file
```

LINK
Statement

Remarks

This command is not used in Compiled or Extended BASIC. Attempting to use it will cause the BASIC Compiler to report a syntax error.

LIST 64

Immediate Mode Command

Format

LIST [linenumber]
LIST [start-finish linenumbers]

Description

The LIST command displays a program or a portion of a program in line number order.

- Display starts at the first line of a program and proceeds to the last line if line numbers are not specified.
- One line is displayed if a single line number is specified. The command is ignored if the line does not exist.
- A portion of a program is displayed if two line numbers are specified. The display will be the lines with numbers between and including the specified lines numbers if they exist.
- If the portion to be listed is larger than one display page (16 lines), the display will scroll upwards until the last line specified has been displayed.
- Use Page Mode and the NEXT PAGE key, or ⟨CTRL⟩/S and ⟨CTRL⟩/Q to stop and restart the display. These functions are discussed in Section 4 of the BASIC manual.

Examples

The following examples illustrate common uses of the LIST command:

COMMAND	RESULTS
`LIST`	Displays the entire program in memory from the first line.
`LIST 500`	Displays only line 500 of the program, if it exists.
`LIST 600-800`	Displays a program segment beginning with the first line after 599 and ending with the last line before 801.

LIST
Immediate Mode Command

Remarks

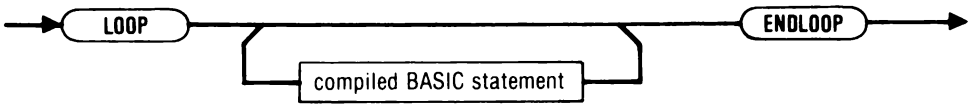
This command is not used in Compiled or Extended BASIC. If you attempt to use this command, the BASIC Compiler will report a syntax error.

LOOP 69A Statement

Usage

```
LOOP
  ...
  statements
  more statements
  ...
ENDLOOP
```

Syntax Diagram



Description

The LOOP statement implements an infinite loop. A LOOP - ENDLOOP pair has the following form:

```
LOOP                ! begin infinite loop
        ...
        ... statements ...
        ...
ENDLOOP
```

The LOOP statement has no explicit termination test; it normally uses the LEAVE statement (described elsewhere) to provide a loop exit.

Control stays within the LOOP - ENDLOOP keywords until a LEAVE statement, GOTO statement, or interrupt is encountered.

Example

The following program fragment repeatedly asks for another value until a zero is entered from the keyboard. The LEAVE statement causes control to leave the loop when the value received is zero.

```
LOOP
        PRINT "Enter another value";
        INPUT a%
        LEAVE IF a% = 0%
        PRINT "Try again...."
ENDLOOP
```

Remarks

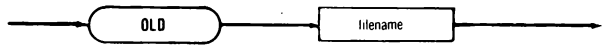
Refer also to the REPEAT and the WHILE statements.

Immediate Mode Command

Usage

OLD {filename}

Syntax Diagram



Description

The OLD immediate mode command is used to load a program into memory from an input, device or file.

The filename, including the optional storage device prefix, filename, and name extension, must be enclosed in quotes.

- OLD may only be used in Immediate mode.
- BASIC will look for the file on the System Device SY0: if a device name is not specified.
- BASIC will look for the file on a specified device if the device name is included as a file name prefix.
- This command assumes that the file named is a valid BASIC program in either ASCII or lexical form.
- If the file name extension is .BAS or .BAL, it does not need to be specified in the file name.
- If no extension is specified, BASIC looks for a file with .BAL name extension and loads that file if it exists.
- If the file named does not exist with a .BAL extension, BASIC looks for the file with a .BAS extension and loads it if it exists.
- If the file exists in both lexical (.BAL) and ASCII (.BAS) form, BASIC will load the lexical form unless the command directly specifies otherwise.
- The OLD command will save the name of a file to be used if a subsequent SAVE, SAVEL, RESAVE, or RESAVEL command is given for which the filename is omitted.

OLD
Immediate Mode Command

Example

The following examples illustrate use of the OLD command:

COMMAND	RESULT
OLD "TEST"	Load the file named TEST.BAL (if present) or TEST.BAS (if TEST.BAL is not present) from the default System Device into memory.
OLD "MF0:TEST.5"	Load the file named TEST.5 from the floppy disk.
OLD "TRAC.TOR"	Load the file TRAC.TOR from the System Device.
SAVEL	Save a lexical version of TRAC.TOR in TRAC.BAL.

Remarks

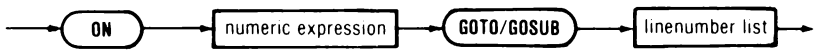
This command is not used in Compiled or Extended BASIC. If you attempt to use this command, the BASIC Compiler will report a syntax error.

ON...GOTO 91
ON...GOSUB
Statement

Usage

ON {expression} GOTO {line list}
ON {expression} GOSUB {line list}
ON {expression} GOTO {statement label list}
ON {expression} GOSUB {statement label list}

Syntax Diagram



Description

The ON GOTO and ON GOSUB statements define multiple control branches. Control transfers to one of the lines listed in the statement, depending on the current value of the expression.

- The expression must be numeric.
- The expression is evaluated and rounded to obtain an integer.
- The integer is used as an index to select a line number from the list contained in the statement.
- The range of the integer must be between 1 and the number of line numbers contained in the list.
- The branch of control may be either a GOTO transfer (refer to the GOTO statement) or a subroutine call (see the GOSUB statement).
- In Compiled and Extended BASIC, the target of the branch of control may be a statement label or a line number.

ON...GOTO
ON...GOSUB
Statement

Example

The following example illustrates a common use of ON-GOTO:

Statement	Meaning
ON A GOTO 400, 500, 600	Transfers control to line 400 if A = 1, to line 500 if A = 2, or to line 600 if A = 3.
ON A GOTO FOO, GFB, PWL	Transfers control to label FOO: if A=1, to label GFB: if A=2, or to label PWL: if A=3. Program statement labels are only allowed in Compiled and Extended BASIC.

The following example illustrates one use of ON-GOTO at line 1030. Since the expression is normally rounded, INT(LOG(R)) is used instead of LOG(R) to ensure that the voltage divider is used for all values less than 1 volt (1000 millivolts).

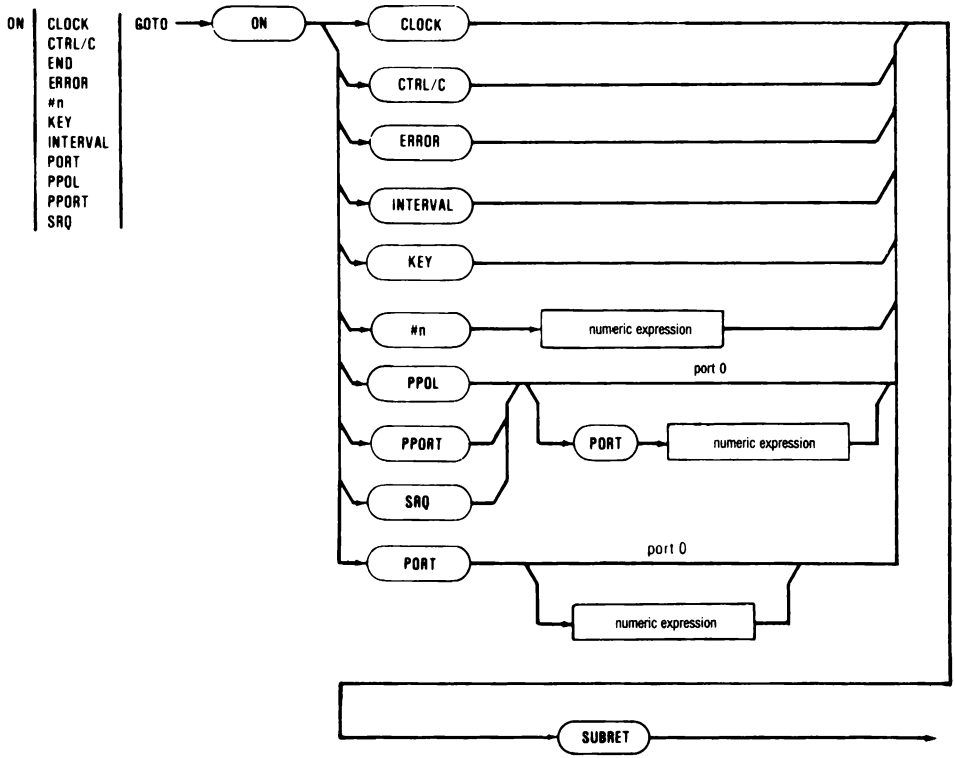
```
1000 REM -- Connect Instrument to Test Station
1010 ! R on input is the value of the voltage to be applied
1020 ! R is in millivolts and is between 10 and 10^6
1030 ON INT(LOG (R)) GOTO 1200,1200,1300,1300,1400,1400
1200 REM -- Setup External Voltage Divider
1210 ! Other statements
1290 GOTO 1500
1300 REM -- Connect Instrument Directly
1310 ! Other statements
1390 GOTO 1500
1400 REM -- Give High Voltage Warning and Then Connect Instrument
1410 ! Other statements
1490 GOTO 1500
1500 REM -- Take Readings
1510 ! Other statements
```

ON-SUBRET 97A Statement

Usage

ON {condition} SUBRET

Syntax Diagram



Description

The SUBRET statement may be used as an instruction for processing an interrupt that occurs while a program is executing a subroutine. When an interrupt occurs, this statement causes control to transfer from the subroutine back to it's calling routine via a SUBRET statement. The interrupt will still be in effect when control returns, which allows interrupt processing via an ON-GOTO statement in the calling routine.

- Any of the available interrupts (ERROR, CTRL/C, etc) may be handled by an ON-SUBRET statement.

ON-SUBRET
Statement

- Any number of ON-SUBRET statements may be cascaded before reaching the point where interrupt processing occurs. This allows several layers of subroutines to return to a main program (if desired) before processing interrupts.

For example, a Main program segment calls subroutine A, which in turn calls subroutine B, which in turn calls subroutine C. An interrupt occurs while subroutine C is in progress. ON-SUBRET statements in subroutines A, B, and C would send control all the way back to the Main segment in turn, where interrupt processing might be performed.

```
!MAIN
      ON CTRL/C GOTO TRAP
       .
       .
       .
CALL A
 .
 .
 .
END

SUB A
     ON CTRL/C SUBRET
     CALL B
SUBEND

SUB B
     ON CTRL/C SUBRET
      .
      .
      .
     CALL C
SUBEND

SUB C
     ON CTRL/C SUBRET
      .
      .
      .
SUBEND
```

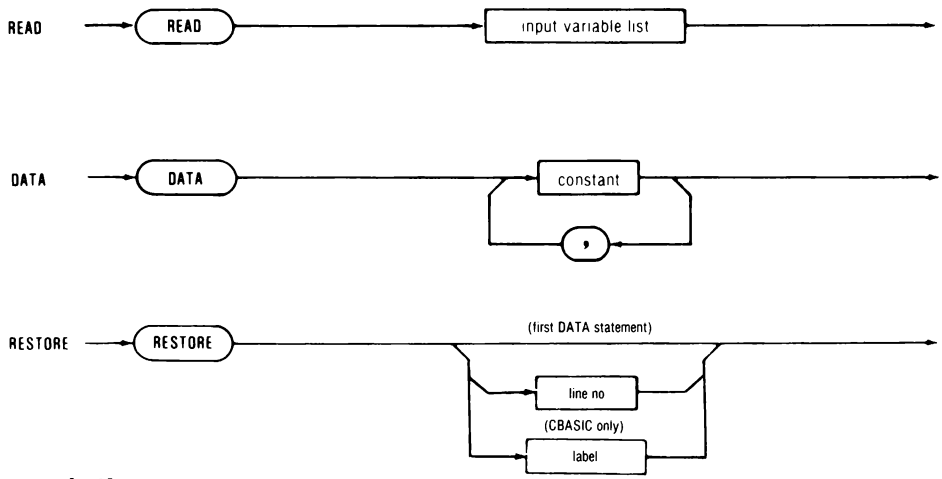
Note that, without the ON-SUBRET statement, an interrupt which occurs in a subroutine will be processed by an interrupt handling statement in the calling routine (if one exists), after which control is returned to the subroutine at the spot where the interrupt occurred. After using an ON-SUBRET statement to respond to an interrupt, the subroutine where the interrupt occurred must be reentered with another CALL statement.

READ, DATA, and RESTORE Statement

Usage

READ {data}
DATA {data list}
RESTORE {linenumber}

Syntax Diagram



Description

The READ, DATA, and RESTORE statements work together as follows:

- The DATA statement defines a sequence of data items to be processed by the READ statement. The data items within a single DATA statement are separated by commas.
- The READ statement assigns data values to a series of one or more variables.
- An array subrange may also be used with the READ statement.

  READ A(0..5)

- The READ statement assigns the next available data items in sequence to the variables referenced.
- The DATA data types must match the corresponding READ variable types (strings for string variables, etc.).

READ, DATA, and RESTORE Statement

- The DATA statement may occur before or after the READ statement.
- The DATA statement must be the last or only statement in the program line. The line may contain no trailing remarks. Everything following DATA is considered to be data.
- Legal data items are: quoted or unquoted strings or numeric constants.
- An unquoted string must not begin with a quote and must not contain commas.
- Leading spaces are ignored unless within a quoted string field.
- Numeric constants may not be quoted.
- A data pointer tracks which data items have been read.
- A DATA statement may contain more items than a subsequent READ statement contains variables.
- A second READ statement may continue reading data (assigning data items to its variables) at the point the first READ statement stopped reading.
- The RESTORE statement resets the pointer to the first data item of the first DATA statement in the program so that the items may be read again.
- If RESTORE specifies a line number, the pointer resets to the first data item of the first DATA statement in or after that program line.
- The RESTORE statement may be executed before all data items have been read from a DATA statement.
- It is not necessary to RESTORE the DATA items if they will be read only once in the program.
- Compiled and Extended BASIC allow a statement label to be used instead of a line number for the RESTORE statement.

Examples

This example illustrates the use of a FOR - NEXT loop to read a list of data items. Each item is a floating-point number, so only one READ statement is necessary. To save each of the data items, use arrays and change line 110 to: READ A(I%). The RESTORE statement in this example would be useful only if the data values were to be used again in the program.

```
10 DATA 1.1, 2.2, 3.3, 4.4, 5.5
20 ! Other statements
100 FOR I%=1% TO 5%
110 READ A
120 PRINT A
130 NEXT I%
140 RESTORE
```

Running this program gives results as follows:

```
1.1
2.2
3.3
4.4
5.5
```

The following example illustrates multiple READ statements and a selective RESTORE statement. Note the double quotes used in the first DATA statement. The double quote is used to allow commas to be inserted in the string data. Note that if the quotes were omitted from line 10, A$ would be TEST FOR HIGH, and line 120 would give an error since the next data element would be LOW, which is not a number. Also note line 410 which resets the data pointer to allow reading the numeric values of line 20. The string data on line 10 is used only once.

```
10 DATA 'TEST FOR HIGH, LOW, AND MEAN VALUES.'
20 DATA 10, 7.5, 9
30 READ A$ \ PRINT A$                          ! Print out heading
100 REM -- Continuously Make Checks
110 ! Check for readings higher than upper limit
120 READ UL                                    ! Get the upper limit
130 ! Other statements
210 ! Check for readings lower than lower limit
220 READ LL                                    ! Get the lower limit
230 ! Other statements
310 ! Compute mean and compare to expected value
320 READ EV                                    ! Get the expected value
330 ! Other statements
400 REM -- Reset instruments and Prepare for Next Test
410 RESTORE 20                                 ! Reset data pointer to UL
420 ! Other statements
500 GOTO 110
```

The program segment that follows causes error 804 (bad DATA format) when statement 110 is executed, since the "!" character is not a legal part of a floating-point number.

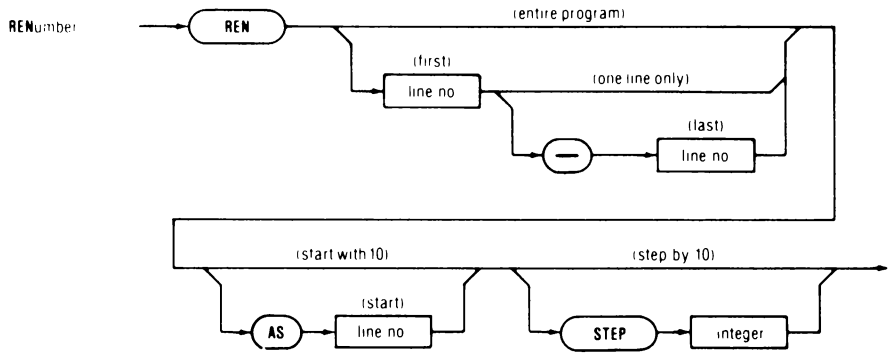
```
100  DATA 1,3,4      ! Configuration data
110  READ A,B,C
```

REN
Immediate Mode Command

Usage

REN
REN [start-stop] AS [new start] STEP [step size]

Syntax



Description

The REN command changes the line numbers of some or all of the program lines in memory. Renumbering is useful to make room for additional program lines.

- REN cannot change the order of program lines.
- REN changes all references to line numbers (i.e., GOTO, GOSUB, etc.) in the program to reflect the new line numbers.
- All items shown in the syntax diagram are optional except the command word REN.
- The entire program is renumbered when no line numbers are specified.
- One line is renumbered when a single line number is specified. The command is ignored if the line does not exist.
- A portion of the program is renumbered when two line numbers are specified.

REN
Immediate Mode Command

- The line number following AS specifies the new starting number of the segment being renumbered. If this would rearrange the sequence of the program, a fatal error occurs and the line numbering remains unchanged.

- When AS is not specified, and a line number or range of line numbers is not specified following REN, the new starting line number is 10.

- When AS is not specified and a line number or range of line numbers is specified following REN, the new starting line number is the same as the old first line number of the range specified.

- The value of the integer expression following STEP must be positive. It defines the difference between any two consecutive, renumbered lines.

- If there is no STEP keyword, the line increment is 10.

- If the value of the integer expression following STEP is so large that the new line numbers would force the program to be rearranged, a fatal error occurs and the lines are not changed.

CAUTION

Renumbering from lower to higher line numbers (with more digits) may cause the renumbered lines to exceed the 79-character maximum line length allowed by BASIC. Restricting program lines to 74 characters maximum length will generally eliminate this problem. This exception is on long lines which include line number references (e.g., ON expression GOTO, IF-THEN-ELSE with line numbers, etc).

NOTE

Program lines containing the ERL (error line) function may have statements such as IF ERL = 200 THEN RESUME 400. The expression, the constant 200, is used as a line number reference. It is not changed during renumbering. It may need to be changed to the correct line number manually.

REN
Immediate Mode Command

Examples

The following program is used in the renumbering examples below:

```
10   A = 1
20   PRINT A + A
30   A = A + 1
40   IF A <= 2 THEN 20
50   PRINT "Done!"
60   END
```

COMMAND	RESULT
`REN 60 AS 32767`	Change line 60 to read: `32767 END`

`REN 10-50 AS 5 STEP 5`	Change lines 10 through 50 to read:
	`5 A = 1`
	`10 PRINT A + A`
	`15 A = A + 1`
	`20 IF A <= 2 THEN 10`
	`25 PRINT "Done!"`
	Note the changed reference in line 20.

`REN`	This would in this case restore the program back to its original form.
`REN 60 AS 1000`	Renumber only line 60 as line 1000. An error results if any lines are numbered between 60 and 1001, since this would rearrange program sequence.
`REN 60 AS 1000 STEP 5`	Same as the previous example. STEP is ignored when only one line is renumbered.
`REN 10-500 AS 1000`	Renumber lines 10 through 500 to start at 1000, in steps of 10.

REN
Immediate Mode Command

Remarks

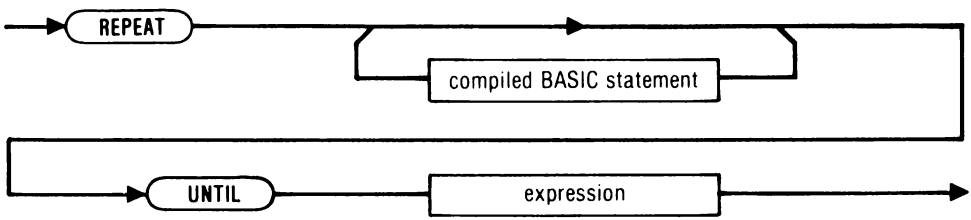
This command is not used in Compiled or Extended BASIC. If you attempt to use this command, the BASIC Compiler will report a syntax error.

REPEAT and UNTIL Statement

Usage

```
REPEAT
  ...
  statements
  more statements
  ...
UNTIL  expression
```

Syntax Diagram



Description

The REPEAT statement introduces a REPEAT - UNTIL loop. Unlike the WHILE loop a REPEAT - UNTIL loop makes its termination test at the bottom of the loop rather than at the top. The REPEAT - UNTIL loop has the following form:

```
REPEAT
        ...
        ... statements ...
        ...
UNTIL {expression}
```

- A REPEAT - UNTIL loop is always executed at least once, since the termination test is not made until the loop has been traversed.
- The action of this loop is to execute the statements in the loop body repeatedly until the value of the expression is true (nonzero).
- An expression is anything that can be evaluated to true or not true.

REPEAT and UNTIL Statement

Example

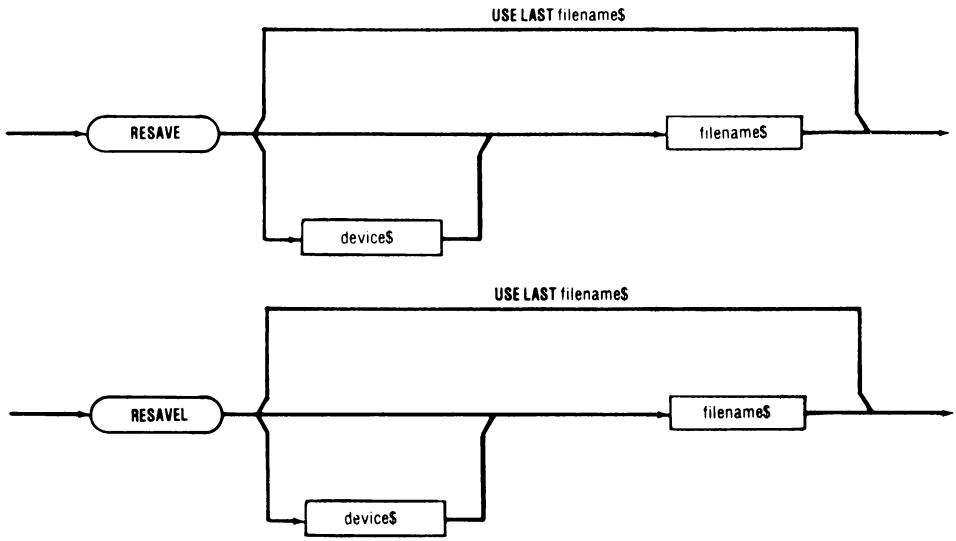
The following loop reads lines of input from channel 2 until a line starting with a letter is found:

```
REPEAT
        PRINT "Reading another line"
        INPUT LINE #2%, a$
        ch$ = LEFT(LCASE$(a$), 1%)
UNTIL "a" <= ch$ AND ch$ <= "z"
```

Usage

RESAVE [filename$]
RESAVEL [filename$]

Syntax Diagram



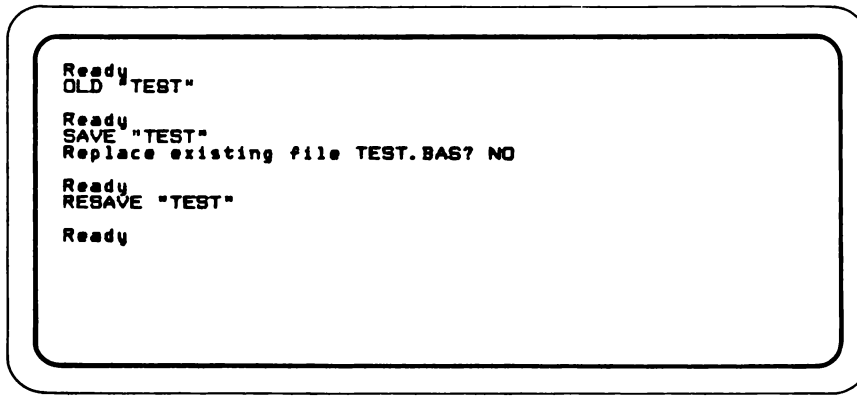
Description

The RESAVE and RESAVEL are an alternative to the SAVE and SAVEL commands. The SAVE and SAVEL commands ask whether or not the user wants to overwrite an existing .BAS or .BAL file. The RESAVE and RESAVEL commands will assume that any existing program file of the same name should be overwritten, and will not ask the user to confirm that the file should be clobbered.

- The RESAVE statement stores an ASCII program. See the SAVE statement.
- The RESAVEL statement stores an lexical format program. See the SAVEL statement.
- The RESAVE statement always uses the default file name extension .BAS. If your program uses a different extension than .BAS, you must specify the entire filename each time the RESAVE statement is used.

- The RESAVEL statement always uses the default file name extension .BAL.

Example



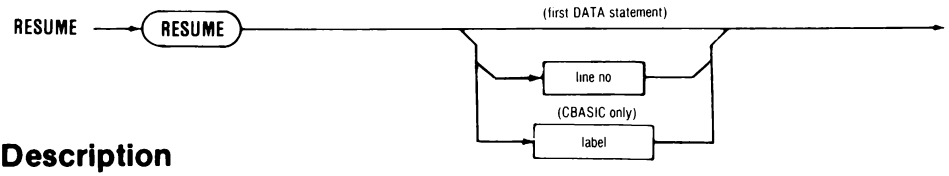
Remarks

This command is not used in Compiled or Extended BASIC. Attempting to use it will cause the BASIC Compiler to report a syntax error. Compiled and Extended BASIC do not have an Immediate mode.

Usage

RESUME [line number]

Syntax Diagram



Description

The RESUME statement acknowledges an interrupt and allows program operation to resume with the next statement after the one being completed when the interrupt occurred or at another specified program location.

- For interrupts other than ON/ERROR interrupts, RESUME (no line number) branches to the statement following the one being executed at the point where the interrupt occurred.

- For ON ERROR interrupts, the Statement causing the error is branched to when no line number is specified.

- If the interrupt occurred in a multiple-statement line, the program resumes with the next statement on the line. There are two exceptions:

  1. Recoverable errors: The program resumes at the beginning of the statement that caused the error.

  2. Input Warning errors 801, 802, and 803: The INPUT statement which caused the error requests the value to be entered again. It did not accept the erroneous entry.

- RESUME (line number) branches to the specified line number.

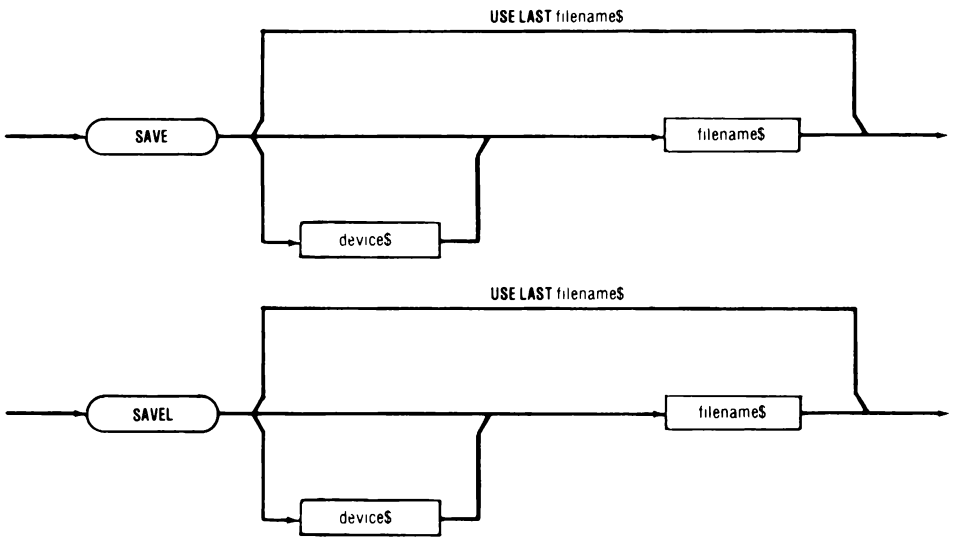
- RESUME [label] branches to the specified statement label in Compiled and Extended BASIC only.

- RESUME terminates the interrupt handler routine.

Usage

SAVE [filename$]
SAVEL [filename$]

Syntax Diagram



Description

The SAVE and SAVEL statements are used to store the user program currently residing in memory as a file. SAVE stores the file in ASCII text form. SAVEL stores the file in lexically analyzed form. A discussion of these forms follows.

- A file name may be specified. It must be enclosed in single or double quotes. The file name may also be specified by a string variable.

- If no file name is specified, the same name will be used that was used in the OLD statement or the RUN statement that was used to load the program into memory.

- If no file name is specified for a new file, an error will occur.

- The file is stored on the default System Device if the file name is not preceded by MF0: for the floppy disk, or ED0: for the optional electronic disk. Refer to the Input and Output Statements section for a discussion of the default System Device concept.

- SAVE adds the file name extension .BAS when an extension is not specified.

- BASIC will request a confirming YES from the keyboard if an attempt is made to store a file under an existing file name. To avoid this interruption in a running program, use the RESAVE statement or first delete the file using a KILL statement.

- SAVE stores the ASCII form of the program in the largest available file space.

- SAVEL stores the lexical form of the program in the first available file space large enough to hold it. Note that this is different from SAVE.

- SAVE may also be used to obtain a printed listing of a program. To do so, specify the device name as either KB1: (RS-232-C Port 1) or KB2: (RS-232-C Port 2). The file name and extension are not required. See the User Manual for details on setting serial port baud rates.

In lexically analyzed form, a user program has binary numbers in place of ASCII character strings to represent line numbers, keywords, and operators. This form occupies less space and eliminates a processing step. Fluke BASIC programs in main memory are always in lexically analyzed form, even during editing. BASIC changes them to ASCII character form when needed for display or for storage by a SAVE statement.

Working copies of user programs should be saved in lexically analyzed form, using SAVEL. In this form, programs will occupy less file storage space and will load into memory quicker.

NOTE

A program saved via SAVEL may not be executable if the version of Fluke BASIC under which it was saved differs from the version under which it is to be executed.

The lexically analyzed form of a program cannot be displayed directly by the Utility program or sent to an external printer. Consequently, backup copies of user programs should be saved in ASCII character form, using SAVE. In this form, different versions of Fluke BASIC will be able to load and interpret the program.

Example

Examples of SAVE and SAVEL used in immediate mode follow:

STATEMENT	RESULT
`SAVE`	Save the program currently in memory, in ASCII character form, on the System device, using the same name that was used to OLD it into memory or RUN it. The default filename extension .BAS is always used.
`SAVE "TEST1"`	Save the program currently in memory, in ASCII character form, on the System Device, under the name TEST1.BAS.
`SAVEL 'MF0:TEST2'`	Save the program currently in memory, in lexically analyzed form, on the floppy disk (MF0:), under the name TEST2.BAL.
`SAVE 'ED0:DATA.T71'`	Save the program currently in memory, in ASCII character form, on the optional electronic disk (ED0:), under the name DATA.T71.
`SAVE "KB1:"`	Send the program currently in memory, in ASCII character form, to RS-232-C Serial Port 1.

Remarks

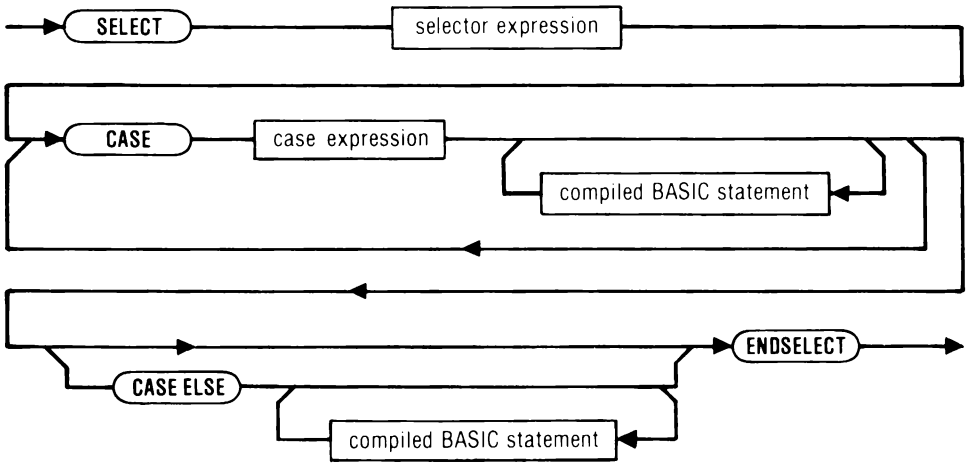
This command is not used in Compiled or Extended BASIC. Attempting to use it will cause the BASIC Compiler to report a syntax error. Compiled and Extended BASIC do not have an Immediate mode.

SELECT 130A Statement

Usage

```
SELECT  selector expression
CASE {case expression}   ! test first condition
...
... statements ...
...
CASE {case expression}   ! test second condition
...
... statements ...
...
CASE {case expression}   ! test next condition, and so on
...
... statements ...
...
CASE ELSE   ! anything not caught above
...
... statements ...
...
ENDSELECT
```

Syntax Diagram



SELECT
Statement

Description

The SELECT statement implements an "n-way" branch, depending upon the value of a "selecting" expression. The SELECT statement is sometimes known as a CASE statement in other languages.

- The {case expression} is a comma-separated list of values formatted as:

  <= expression
  >= expression
  <> expression
  = expression or expression
  < expression
  > expression
  expression
  expression 1 .. expression 2

- The form:

  CASE expression 1 .. expression 2

  says that if the value of the selector expression falls between the values expression 1 and expression 2, that is:

  expression 1 <= selector expression <= expression 2

  then the corresponding CASE should be executed.

- The {selector expression} may be either a number or a string.

- The space CASE clauses simply indicate the values of the {selector expression} for which the corresponding piece of code should be executed. The tests for the different CASEs are performed in the same order as their appearance in the program until one of the CASEs matches the value required. At this point, the code following the CASE is executed until the next CASE occurs; then control flows to the ENDSELECT statement; the remaining CASEs are not evaluated.

- An empty CASE clause (a CASE without any statements following) is also allowed. This permits an explicit "do nothing" CASE to weed out selector values for which no action should be performed. This is illustrated in the example.

- Any of the CASE expressions may involve variables.
- The SELECT statement also permits selector expression to be an integer, floating-point, or string value.
- An "illegal mode mixing" error is reported if the value of the selector and case values are not compatible (i.e., number compared with string).

Example

The following program fragment illustrates the use of the SELECT statement.

```
while (len(string$) (= 5% and eof% = 0%)
        char% = getchar(channel%)
        select (char%)

        case 13%                            ! carriage return
                leave                       ! leave WHILE loop

        case 26%                            ! EOF
                eof% = -1%                  ! signal end of file

        case 32%, 9% .. 10%                 ! space, tab, linefeed
                ! do nothing

        case ( 32%, )= 127%                 ! other control character
                print "Funny character"; char%; "encountered"

        case 48% .. 57%, 65% .. 90%         ! digit or uppercase
                string$ = string$ + chr$(char%)

        case 97% .. 122%                    ! lowercase letter
                string$ = string$ + ucase$(chr$(char%))

        case else                           ! other punctuation

                punct$ = punct$ + chr$(char%)

        endselect

        !
        ! select course of action from short option list
        !
        select (string$ + punct$)

        case a$ + "!"
                print "Quick mode"

        case "end"
                leave

        case "quit"
                exit

        endselect
endwhile
```

STEP 144A
Immediate Mode Command

Format

STEP

Description

The Immediate Mode STEP command sets a mode in which each statement within a program is executed individually by pressing RETURN.

- STEP must first be enabled by a breakpoint stop in a running program, caused by STOP ON or CONT TO.
- After a breakpoint stop, type STEP to select Step Mode.
- From Step Mode, type ⟨CTRL⟩/C or any Immediate Mode command to return to Immediate Mode.
- Any BASIC command or statement that is available in Immediate Mode can also be used to exit Step Mode.
- In Step Mode, one statement is executed each time RETURN is pressed.
- After executing each statement, the display reads: STOP ON LINE n, where n is the next line to be executed.
- When used with a variable TRACE ON, the display will also show changes in selected variables whenever a statement assigns a new variable value.

Remarks

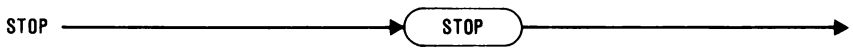
This command is not used in Compiled or Extended BASIC. Attempting to use it will cause the BASIC Compiler to report a syntax error. Compiled and Extended BASIC do not have an Immediate mode.

Statement

Usage

STOP

Syntax Diagram



Description

The STOP statement halts execution of the BASIC program and displays the line number where the STOP occurred.

- This is the only form of the STOP statement allowed by Compiled and Extended BASIC.
- STOP terminates program execution.
- STOP can be used to indicate "dead end" code branches, either because of errors or because of logical structuring.

Example

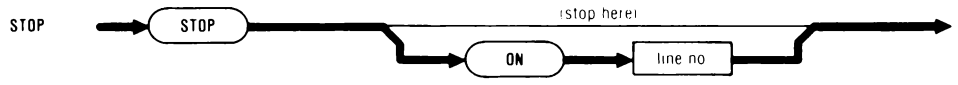
The following example illustrates a common use of STOP:

```
10 REM -- TEST PROGRAM
20 ! Other statements
30 !
999 STOP                          ! End of main procedure
1000 REM -- SUBROUTINES
1010 ! Other statements
5000 END
```

Usage

STOP ON {line number}

Syntax Diagram



Description

The STOP ON statement stops execution of a program.

- STOP ON line number, allows a program to be run in sections during logic debugging.
- The program stops at the line number of the STOP when ON line number is not included.
- The program stops at the line number following ON, without executing it, when ON line number is included.
- STOP ON may be executed in either Immediate or Run Mode.
- STOP ON line number enables the STEP command.

Remarks

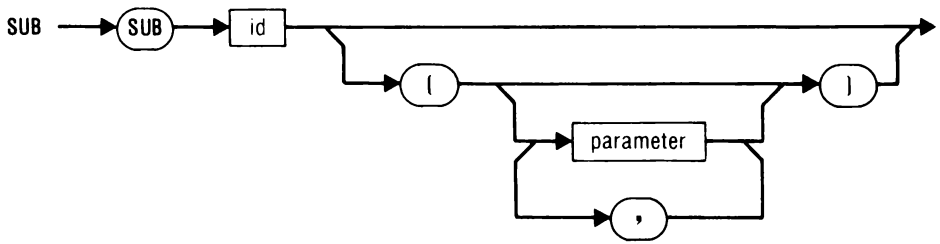
This command is not used in Compiled or Extended BASIC. Attempting to use it will cause the BASIC Compiler to report a syntax error. Compiled and Extended BASIC do not have an Immediate mode.

SUB 149A Statement

Usage

SUB {subroutine name}(parameter list)

Syntax Diagram



Description

The SUB statement marks the beginning of a true Compiled or Extended BASIC subroutine. All statements between the SUB statement and a SUBEND statement (see definition below) constitute a Compiled or Extended BASIC subroutine. The SUB statement also defines a list of parameters that will be passed to that subroutine from the calling routine.

- The SUB statement must be the first statement on a program line.
- Parameters may be passed from a calling routine by value (as an expression) or by reference (by name, such as a variable or an entire array name).

An example of a SUB statement is

```
100 SUB DVM(READINGS%(), MESSAGE$, SCALE)
```

which identifies the beginning of a subroutine named DVM that will exchange three parameters with a calling routine. The parameters are an integer array READINGS%(), a string called MESSAGE$, and a floating-point parameter named SCALE. All of the statements between this SUB statement and the next following SUBEND statement are part of the subroutine DVM.

SUB
Statement

This sample subroutine will be used via a CALL statement like this:

```
100 CALL DVM(A%(),B$,C)
```

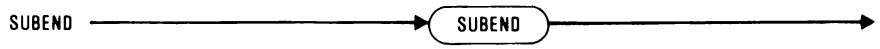
When the subroutine DVM runs, it will use the variable array A%() out of the calling routine, only with the local name of READINGS%(). B$ From the calling routine will be used in the subroutine as MESSAGE$, and C will be used as SCALE.

SUBEND 149B Statement

Usage

SUBEND

Syntax Diagram



Description

The SUBEND statement marks the end of a true subroutine. All of the statements occurring between this statement and the last previous SUB statement constitute a subroutine.

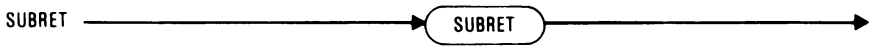
- Only one SUBEND statement may be used with each SUB statement.
- The SUBEND statement returns program control to the routine that called the subroutine.

SUBRET 149C Statement

Usage

SUBRET

Syntax Diagram



Description

The SUBRET statement returns control from a subroutine back to the calling routine. It is used to return to a calling routine from places other than at the end of a subroutine.

- Any number of SUBRET statements may be used in a subroutine.
- The SUBRET statement must be used within the body of a subroutine.
- The SUBRET Statement is analogous to the RETURN statement in BASIC subroutines that are entered with a GOSUB statement.

SYSTEM 150 Variables

Description

System variables store changing event information for use as required by a program. They are accessed by name and return a result in floating-point, integer, or string form as appropriate. The table below lists the system variables and gives their meaning and form.

System Variables

NAME	TYPE	EXAMPLE	MEANING
DATE$	String	08-Feb-81	Current date in the format DD-MM-YY.
ERL	Integer	1120	Line number at which the most recent BASIC program error occurred.
ERR	Integer	305	Error code of the most recent error the BASIC interpreter found in the program being executed.
ERR$	String	$MAIN$	Name of subroutine where error occured.
FLEN	Integer	6	Length of the last file opened in 512-byte blocks.
KEY	Integer	20	Position number of the last Touch-Sensitive Display region pressed.
MEM	Floating-Point	29302	Amount of unused main memory, expressed in bytes.
RND	Floating-Point	0.2874767	Pseudo-random number greater than 0 and less than 1. Repeatable if not preceded by RANDOMIZE.

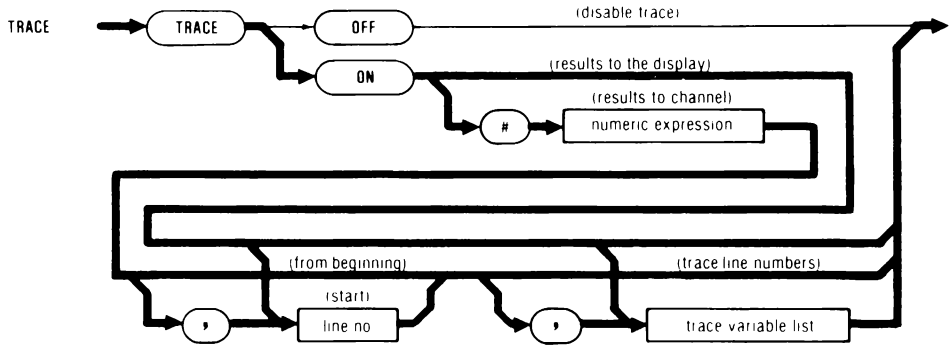
SYSTEM
Variables

TIME	Floating-Point	0.5491405E+08	Number of milliseconds since the previous midnight.
TIME$	String	17:45	Current time of day in 24-hour format.
STIME$	String	17:45:19	Current time of day, including seconds.
CMDFILE	Integer	-1	Command file active.
CMDLINE$	String	BASIC	Current Operating System command line.

IBASIC Usage

TRACE ON
TRACE ON [linenumber][trace variable list]
TRACE ON [channel#n][linenumber][trace variable list]

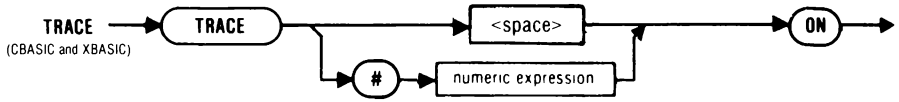
Syntax Diagram



CBASIC and XBASIC Usage

TRACE ON

Syntax Diagram



Description

TRACE prints a record of line numbers encountered or changes in variable values.

- If a previously opened channel is specified, the results of the trace are sent to the channel. Otherwise, the results are sent to the display.
- A starting line number for tracing may be specified. If it is not specified, tracing starts with the first line following the execution of the TRACE statement.
- Tracing is activated when the specified start line or the first line is encountered.

TRACE ON
Statement

Line Number Tracing

- TRACE may be used in either Immediate or Run Mode.

- The only form of the TRACE statement allowed in Compiled and Extended BASIC is listed in the CBASIC and XBASIC Usage syntax diagram. Both CBASIC and XBASIC permit line number traces to be sent to a channel, but variable tracing is not allowed.

A line number trace has the following forms:

STATEMENT	MEANING
`TRACE ON`	Trace line numbers from the first line and send the results to the display.
`TRACE ON line number`	Trace line numbers from the specified line and send the results to the display.
`TRACE ON # channel`	Trace line numbers from the first line and send the results to the open channel.
`TRACE ON # channel, line number`	Trace line numbers from the specified line and send the results to the display.

- A line number trace and a variable trace will not execute concurrently.

- A line number trace occurring after a variable trace specifies a new line number after which variable tracing will resume, provided no TRACE OFF occurred in the interim.

TRACE ON
Statement

Example

The following examples illustrate the results of different forms of line number trace statements.

STATEMENT		RESULT
30	TRACE ON	Start a line number trace at the next line following line 30.
500	TRACE ON 1275	Start a line number trace when line 1275 is reached.
750	TRACE ON # 3%, 400	Start a line number trace when line 400 is reached. Send the trace output to channel 3.

The line number trace displays a series of numbers representing the line numbers or the statements executed. The following example illustrates typical results.

Program

```
10      TRACE ON
20      I% = 0%
30      I% = I% + 1%
40      IF I% < 3% THEN 30
50      TRACE OFF
60      END
```

Results

```
20
30
40
30    Showing that the loop was
40    executed 3 times
30
40
50

Ready
```

TRACE ON
Statement

Variable Tracing

A variable trace has the following forms:

STATEMENT	MEANING
`TRACE ON variable list.`	Trace changes in value of selected variables from the first line, and send the results to the display.
`TRACE ON #channel, variable list`	Trace changes in value of selected variables from the first line, and send the results to the open channel.
`TRACE ON line number, variable list`	Trace changes in value of selected variables where the specified line is encountered, and send the results to the display.
`TRACE ON #channel, line number, variable list`	Trace changes in value of selected variables from the specified line, and send the results to the open channel.

- If a list of variables is specified, the trace is of changes in values of those variables. Otherwise, the trace is of line numbers encountered.
- A variable trace may specify one or more variables of any type: string, integer, and floating point.
- A variable trace of an array may use the form A() as the variable. A() means "TRACE ON all elements of array A".

- An array must be previously dimensioned before tracing.
- A variable trace and a line number trace will not execute simultaneously.
- Two or more variable traces will execute simultaneously. For example, TRACE ON A followed by TRACE ON B is equivalent to TRACE ON A,B.
- A variable trace occurring after a line number trace turns off the line number trace.

A variable trace statement resembles the line number trace statement except that a list of variable names is included. The following example specifies trace output to channel 2, tracing to start at line 340, and tracing of changes in values of A%, element (3,4) of array B, and all of array A$:

```
30      TRACE ON #2%, 340, A%, B(3%, 4%), A$()
```

NOTE

A variable trace of an array cannot be done without first dimensioning the array with a DIM statement.

Variable trace display output takes the following form:

line number identifier type(indices) = new value

Where:

1. Line number is the number of the line in which the variable was assigned a new value.
2. Identifier is the name of the variable.
3. Type is % for integers, $ for strings.
4. Indices identify which element of the array is being traced on and displayed (for array elements only).
5. New value is the new value assigned.

TRACE ON
Statement

This example shows the result of a trace of an array variable:

```
TRACE ON A (1, 2)                    Displays the value of
                                     A(1,2) when it is assigned.
                                     For example,

                                     220 A(1,2) = 47.3386
```

This example program illustrates the display resulting from a trace of an integer array program:

```
10      DIM A% (2%, 2%)
20      TRACE ON A%()
30      FOR I% = 0% TO 2%
40      FOR J% = 0% TO 2%
50      A% (I%, J%) = I% * J%
60      NEXT J%
70      NEXT I%
80      TRACE OFF
90      END
```

Results

```
50  A%(0,0) = 0
50  A%(0,1) = 0
50  A%(0,2) = 0
50  A%(1,0) = 0
50  A%(1,1) = 1
50  A%(1,2) = 2
50  A%(2,0) = 0
50  A%(2,1) = 2
50  A%(2,2) = 4
```

Other Trace Options

TRACE ON line number can be used to define a trace region within a program. The example below traces the array A$ only in the subroutine starting at line 110. Until TRACE OFF is executed, TRACE ON continues to trace all variables for which a TRACE ON was executed, and continues to send trace output to the specified channel or the display.

```
10      DIM A$ (5%, 5%)
20      TRACE ON 110, A$()              ! Start tracing array A$ at
                                        line 110
30      !
40      FOR I% = 0% to 5%
50      A$ (I%, 0%) = CHR$ (ASCII (' ') + I%)
60      GOSUB 110
70      NEXT I%
80      TRACE OFF
90      STOP
100     !
110     FOR J% = 1% TO 5%
120     A$ (I%, J%) = A$ (I%, J% - 1%) + CHR$ (ASCII(' ') + I%
        + J%)
130     NEXT J%
140     TRACE ON 110
150     RETURN
160     END
```

The following trace display output results from running this program. Refer to Appendix I, ASCII/IEEE-1978 Bus Codes, and note the display characters that follow SPACE, character number 32, for clarification of these results.

```
120     A$(0, 1) = !
120     A$(0, 2) = !"
120     A$(0, 3) = !"#
120     A$(0, 4) = !"##
120     A$(0, 5) = !"##%
120     A$(1, 1) = !"
120     A$(1, 2) = !"#
120     A$(1, 3) = !"##
120     A$(1, 4) = !"##%
120     A$(1, 5) = !"##%&
120     A$(2, 1) = "#
120     A$(2, 2) = "##
120     A$(2, 3) = "##%
120     A$(2, 4) = "##%&
120     A$(2, 5) = "##%&'
120     A$(3, 1) = ##
120     A$(3, 2) = ##%
120     A$(3, 3) = ##%&
120     A$(3, 4) = ##%&'
120     A$(3, 5) = ##%&'(
120     A$(4, 1) = #%
120     A$(4, 2) = #%&
120     A$(4, 3) = #%&'
120     A$(4, 4) = #%&'(
120     A$(4, 5) = #%&'()
120     A$(5, 1) = %&
120     A$(5, 2) = %&'
120     A$(5, 3) = %&'(
120     A$(5, 4) = %&'()
120     A$(5, 5) = %&'()*
```

It is also possible to send trace output to different channels. Output is sent to one channel at a time. The following example illustrates this:

```
10      OPEN "TRACE1.DAT" AS NEW FILE 1%! First trace channel
20      OPEN "TRACE2.DAT" AS NEW FILE 2%! Second trace channel
100     TRACE ON A%, B$(), C()          ! Send output to console
250     TRACE ON #1%                    ! Send output to channel 1
370     TRACE ON #2%                    ! Send output to channel 2
1010    TRACE OFF                       ! Discontinue all tracing
1020    END
```

TRACE ON
Statement

TRACE OFF Statement

TRACE OFF disables any pending or active trace assigned in the program and destroys the variable list.

The following example illustrates that TRACE ON starts only a line number trace after TRACE OFF:

```
10      TRACE ON A, B                   ! Trace variables A and B
50      TRACE OFF                       ! Halt the variable trace
100     TRACE ON                        ! Start a line number trace
```

The following example illustrates a way to suspend tracing until a later point in a program:

```
10      TRACE ON A, B                   ! Trace variables A and B
50      TRACE ON 100                    ! Stop trace until line 100
100     ! Resume tracing variables A and B
```

UNLINK 160 Statement

Usage

UNLINK

Syntax Diagram

Description

The UNLINK statement removes all reference to Assembly Language or FORTRAN subroutines from memory, making the previously reserved memory space available for other uses. Individual Assembly Language or FORTRAN subroutines cannot be selectively removed from memory.

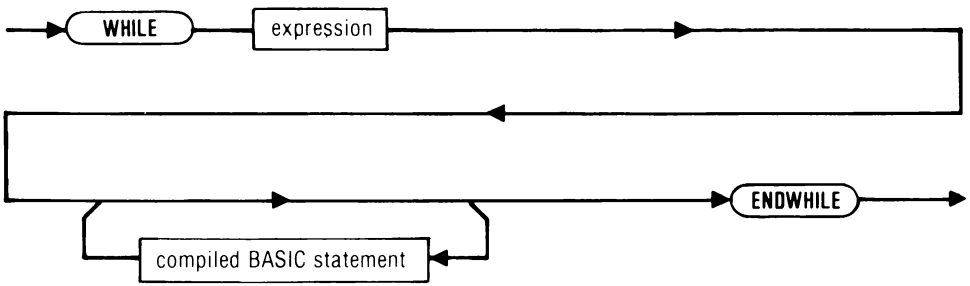
- Memory freed by UNLINK is available for other uses.
- UNLINK can be used in either the Immediate or the Run mode.
- UNLINK cannot be used in Compiled or Extended BASIC programs.

WHILE 170 Statement

Usage

```
WHILE {expression}
  ...
  statements
  more statements
  ...
ENDWHILE
```

Syntax Diagram



Description

The WHILE statement introduces a WHILE loop. Its form is:

```
WHILE {expression}
        ...
        ... statements ...
        ...
ENDWHILE
```

The {expression} is rounded to integer and evaluated. While the expression's value is true (nonzero), the loop surrounded by the keywords WHILE and ENDWHILE is executed.

- The expression is evaluated at the top of the loop, thus the loop is not executed if the expression evaluates to not true.
- An expression is anything that can be evaluated as true or not true.

WHILE
Statement

Example

This program fragment calls a subroutine named GETVAL as long as the value returned by the subroutine is less than 105% or greater than 240%.

```
WHILE i% ( 105% OR i% ) 240%
        CALL getval(i%)
        PRINT "The new value is"; i%
ENDWHILE
```

Remarks

The WHILE statement evaluates its loop control statement at the beginning of the loop. The REPEAT - UNTIL statement evaluates its loop control statement at the end of the loop.