M30 M40

BASIC Language
User Guide

olivetti L1

M30 M40

BASIC Language
User Guide

olivetti L1

PREFACE

This manual is a user guide for programmers and analysts working on BASIC application program development for the commercial sector.

It assumes a prior knowledge of basic EDP concepts and of high level programming languages.

The reading of "Introduction to MOS" and "BASIC Programming Environment-Features and Functions" is a prerequisite for the correct use of this manual.

SUMMARY

The manual describes the BASIC syntax and semantic characteristics.

It is divided into two main parts:

- Part 1 is a BASIC programming guide for the M30/M40 system. It examines the language characteristics and describes the use of BASIC statements which are grouped by function.

- Part 2 gives an indepth description of each BASIC statement, command or function. These are presented in alphabetical order.

REFERENCES

Read first...

Introduction to MOS
Code 4002130 G

BASIC Language Programming Environment - Features and Functions
Code 4002530 Q

For further information, read...

BASIC Language - Graphic Handling User Guide
Code 4004470 B

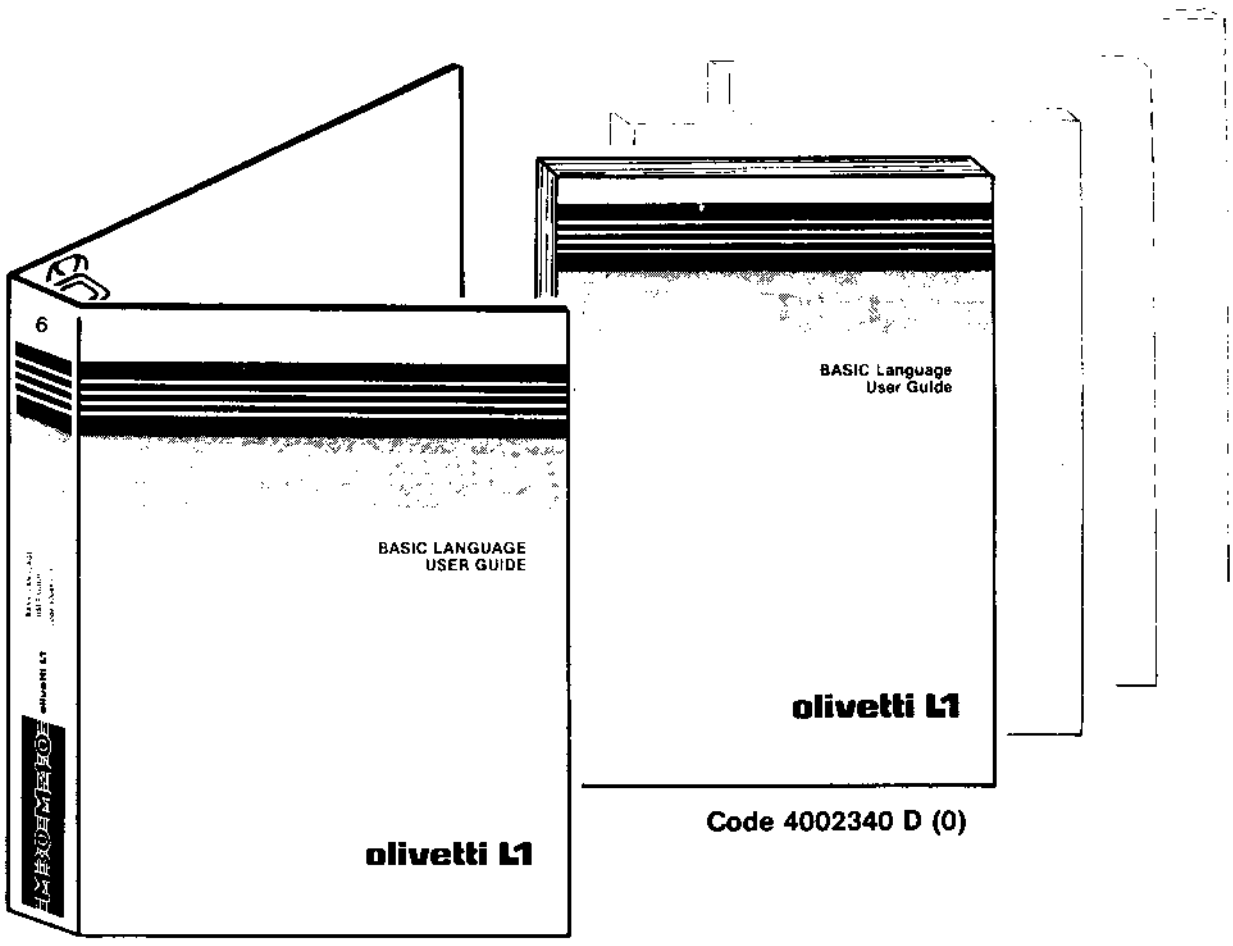
MOS - EDITOR Reference Manual
Code 4002440 P

Glossary/Glossario
Code 4002140 H

First Edition: January, 1983

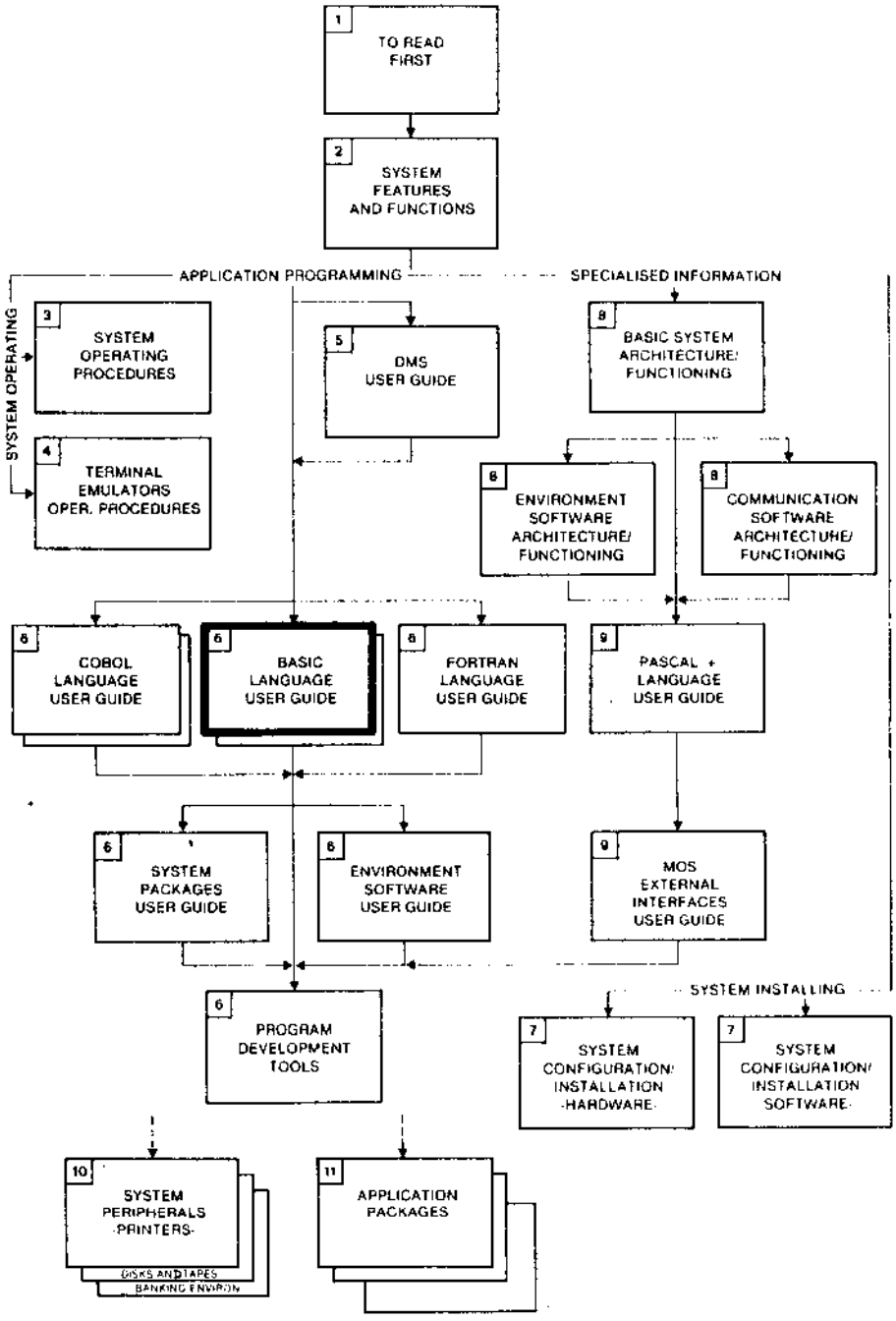
PUBLICATION ISSUED BY:

Ing. C. Olivetti & C., S.p.A.
Servizio Centrale Documentazione
77, Via Jervis - 10015 IVREA (Italy)



Code 4004830 Y

Code 4002340 D (0)


1
TO READ FIRST
2
SYSTEM FEATURES AND FUNCTIONS
APPLICATION PROGRAMMING
SPECIALISED INFORMATION
SYSTEM OPERATING
3
SYSTEM OPERATING PROCEDURES
4
TERMINAL EMULATORS OPER. PROCEDURES
5
DMS USER GUIDE
8
BASIC SYSTEM ARCHITECTURE/ FUNCTIONING
8
ENVIRONMENT SOFTWARE ARCHITECTURE/ FUNCTIONING
8
COMMUNICATION SOFTWARE ARCHITECTURE/ FUNCTIONING
6
COBOL LANGUAGE USER GUIDE
6
BASIC LANGUAGE USER GUIDE
6
FORTRAN LANGUAGE USER GUIDE
9
PASCAL + LANGUAGE USER GUIDE
6
SYSTEM PACKAGES USER GUIDE
6
ENVIRONMENT SOFTWARE USER GUIDE
9
MOS EXTERNAL INTERFACES USER GUIDE
SYSTEM INSTALLING
6
PROGRAM DEVELOPMENT TOOLS
7
SYSTEM CONFIGURATION/ INSTALLATION -HARDWARE-
7
SYSTEM CONFIGURATION/ INSTALLATION SOFTWARE-
10
SYSTEM PERIPHERALS -PRINTERS-
DISKS AND TAPES
BANKING ENVIRON.
11
APPLICATION PACKAGES

1. GENERAL INFORMATION

In this chapter the BASIC language is defined and a description of the main characteristics is given for BASIC on the M30/M40 system.

The BASIC Language

BASIC (Beginner's All-Purpose Symbolic Instruction Code) is a general purpose, high level programming language. You can use BASIC to solve both business and scientific problems. BASIC consists of self-explanatory statements and commands.

BASIC is an interactive language with frequent operator-system communication during program execution and debugging.

Main Characteristics of BASIC on the M30/M40 System

The BASIC language on the M30/M40 system can be used to solve business and also scientific problems as it includes some mathematical functions and statements that can be used to handle very complex numeric expressions. BASIC provides you with:

- powerful and simple handling of external files
- program segmentation using chaining
- a large number of built-in or system functions
- handling of character strings through chaining and built-in string functions
- handling of arrays (up to 10 dimensions)
- simple but powerful control of the print format
- simple but powerful video and keyboard management

MODES OF OPERATION

BASIC provides you with three different modes of operation: COMMAND MODE, EXECUTE MODE and EDITOR MODE.

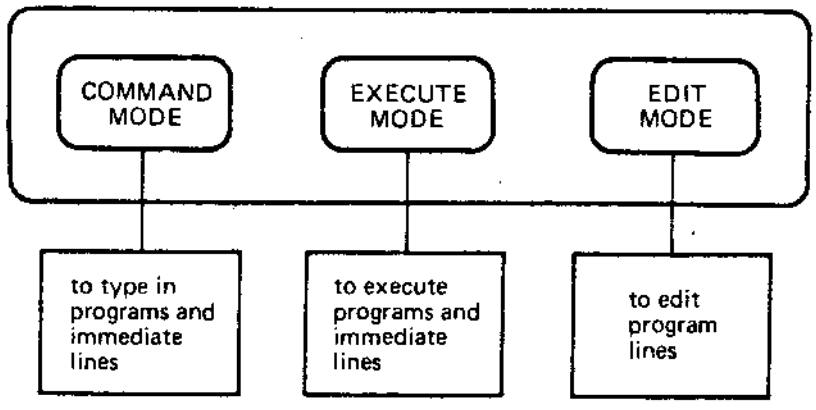


Fig. 1. 1 - Modes of Operation

Command Mode

Whenever the system enters Command Mode, it displays a special prompt:

OK

In Command Mode, BASIC does not accept your input until you complete the line by pressing the carriage return/line feed key.
BASIC always ignores leading spaces in a line. It jumps ahead to the first non-space character. If this character is not a digit, BASIC treats the line

as an immediate line. If it is a digit, BASIC treats the line as a program line. An immediate line contains one or more BASIC statements separated by a colon, and ending with a carriage return/line feed. A program line consists of a line number (from 0 to 65529), one or more BASIC commands or statements (separated by colons) ending with a carriage return/line feed.

Command Mode includes the following Submodes:

- Immediate (or Direct) when you enter an immediate line
- Program (or Indirect) when you enter a program line.

Execute Mode

The M30/M40 executes BASIC statements or commands in Execute Mode. A BASIC program is executed in ascending line number sequence unless a control statement (WHILE/WEND, ON...GOTO, IF...THEN ... ELSE, FOR/NEXT, GOTO) dictates otherwise.

Line Editor Mode

BASIC includes a Line Editor for correcting program lines. This is useful for correcting long and complex lines without having to re-enter them completely.

The system enters Editor Mode when

- the EDIT command is entered
- a syntax error is detected

The system will then display the line to be edited and will wait until the user enters an Editor command .

Changing Mode or Environment

The operation mode (Command, Editor or Execution Mode) or environment (SHELL or BASIC) may be changed by entering certain commands or control characters or if certain conditions occur.

The table below summarizes how you can change mode or environment.

IF the system is in..	AND IF...	THEN...
Execute mode	You press /CTRL/ /C/ while the system is executing a BASIC program or an immediate line	Execution is interrupted and the system enters BASIC Command Mode
	a syntax error is detected	The system enters BASIC Line Editor Mode
STOP,STON command execution	the execution of a BASIC program or command is completed OR an error other than a syntax error is detected	The system enters BASIC Command Mode
BASIC Command Mode	you enter an immediate line	The system enters BASIC execution mode
	you enter SYSTEM SYSTEM can also be used in a BASIC program	The System enters SHELL and user memory is cleared
	you enter EDIT nn..n OR EDIT.	The system enters BASIC Line Editor Mode
SHELL	you enter BASIC	The system enters the BASIC environment (in Command Mode)

CHARACTER SET

The BASIC language character set consists of the following:

- alphanumeric characters
- numeric characters
- special characters
- control characters
- function keys
- backspace, new line

Alphanumeric Characters

The alphanumeric characters of the BASIC language consist of upper and lower case letters of the English alphabet.

Numeric Characters

The numeric characters are the digits from 0 to 9.

Special Characters

The special characters available are the following:

ø ; = + - * / ^ () % # $![ ]' : & ? < > / @
_ "

Control Characters

The following control characters are available in the BASIC language:

/CTRL/ /C/: Suspends program execution and causes the system to return to Command Mode.

/CTRL/ /O/: Suspends output on the screen and on the printer whilst execution continues.
The output process can be restored by re-pressing /CTRL/ /O/.

/CTRL/ /S/: Suspends program execution. Execution is resumed by pressing any key.

Function Keys

The function keys of the M30/M40 system are classified as follows:

- special function keys
  (→, ←, |←, →|, RESET, IC, DC, ↖, ↑, ↓)
- end of output keys.

The /BS/ key is also available. This key deletes the last character that has been input.

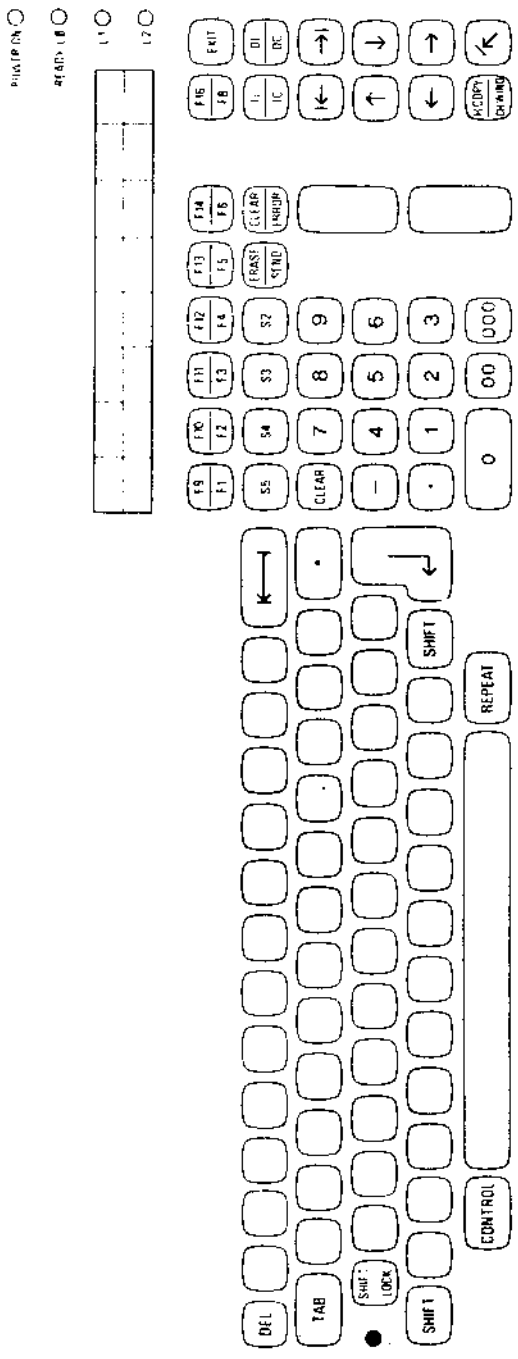


Fig. 1. 2 - Business Keyboard

2. NOTATION CONVENTION AND LEXICAL RULES

This chapter describes the syntax adopted by BASIC and the relative lexical rules.

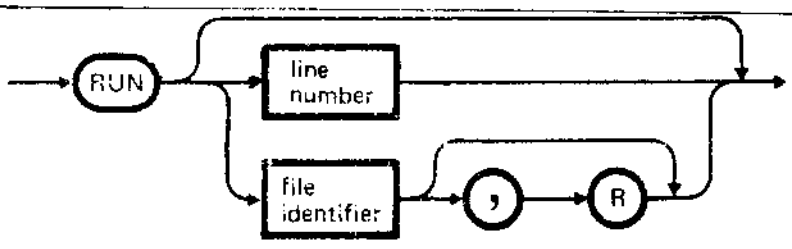
Syntax

The syntax used to describe the structure of a BASIC command, statement or function is based on syntax diagrams.

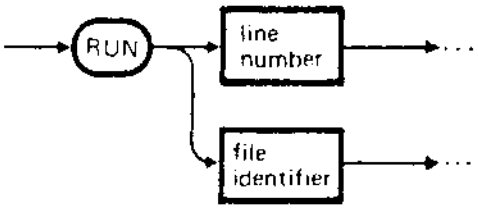
A syntax diagram is a flow-chart with one entry and one exit. Each path through the diagram defines an allowable sequence of symbols. Given below are the rules you must follow to draw a syntax diagram:

- all items enclosed by a rounded envelope (ovals and circles) must be entered exactly as shown
- all items enclosed in a rectangular box are the names of parameters used in statements, commands, or functions
- a description of each parameter is given in the text following the drawing (when it is not interpreted immediately).

For example:

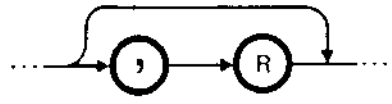


- a fork indicates a choice. You must select one path.

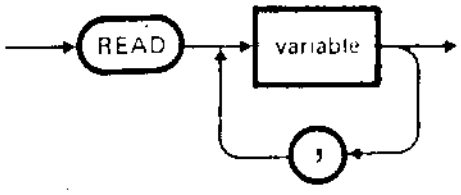


In the example,after RUN you may either enter a line number or file identifier.

- a branch without parameters indicates that the alternative is a by-pass (see the following example for the path which exceeds ,R)



- a loop indicates a repetition. For example, the parameter "variable" may be repeated n times in a READ statement and each "variable" must be separated from the next by a comma



- this manual shows BASIC reserved words in upper case letters even though you may enter them in lower case letters.

Program Elements

A BASIC program is made up of a set of statements.

You can enter lines with one or more statements. In the latter case each statement must be separated by a colon (:).
In a BASIC program each line starts with a line number: an integer greater than or equal to 0 or less than or equal to 65529 and ends when you press the carriage return/line feed key. You can enter up to 255 characters per (logical) line. A logical line can include several physical lines.

For example:

```
20 INPUT "Length";B:IF B<=0
                    THEN 20
```

is one logical line divided into two physical lines.

Each BASIC statement contains:

a line number - a keyword - the statement body.

For example:

1010 PRINT A + B

1010 is the line number, PRINT is the keyword and A + B is the statement body

Some statements contain more than one keyword or statement body.

For example:

320 PRINT USING 200,A,B,C

PRINT and USING are keywords, while 200,A,B, and C are the statement bodies

The various element types that make up a statement must be separated by at least one blank.

For example, the following statement,

100 PRINT A

will not be accepted if you write it as follows:

100 PRINTA

Only the two keywords GO TO and GO SUB can be written without blanks after the word GO. So you can write: GOSUB and GOTO

Line Number

The line number must be a positive integer (from 0 to 65529). You must enter the line number as it distinguishes one program line from another. Program line numbers are ordered in memory in ascending line number sequence, irrespective of the order in which they are entered.

It is conventional to use an interval of 10 between each line number. This allows you to insert new program lines between the existing ones.

You can ask the system to number your lines for you through the use of the AUTO command. You can also change the order of the line numbers by altering the first line number and selecting a different increment between two numbers. You do this with the RENUM command.

Keywords

Each statement begins with a keyword (or reserved word). The keyword is a mnemonic of an English word. For syntax reasons it must be preceded or followed by at least one blank.
The keyword defines the type of statement to be carried out. One or more operands (constants or variables) or expressions can be entered after the keyword.
Some statements have more than one keyword, e.g. IF...THEN. You can enter BASIC keywords in lower-case or upper-case letters. They are converted into upper-case letters when listing the program.
Besides keywords, other reserved words are BASIC command names (e.g. LOAD, RUN etc.) or statement names or functions names (e.g. SIN, COS, etc).

Statement Body

The actual statement contains operators and operands, preceded by the keyword of the statement.

You can write some operators and/or operands before the keyword.

For example:
100 if A > B THEN 200

Operators

The operators consist of one or more special symbols.

- the same operator can have a different meaning depending on the statement in which it is used. For example, the equals sign (=) can be used to assign a value (in the LET statement, for example) or it can be a relational operator (in a logical expression).

- some operators are not used with any particular statement and have no pre-established position inside the statement itself.

For example:

```
10 A = (B+C)/D
20 IF A>+A1 THEN 100
```

"(","+",")","/" are examples of these operators.

- some operators are used with a particular statement and have a pre-defined position inside the statement

For example:

```
1000 A = A+B/C
```

"=" is an example of this type of operator.

Operands

You can use the following operands in a BASIC statement:

- simple variables, array elements, arrays
- control items
- numeric constants (i.e. an integer, a fixed decimal point number, a floating decimal point number)
- a string constant
- a hexadecimal constant
- any string of ISO characters in the REM statement
- a file identifier in a statement that refers to an external file.

Blanks

You can insert blanks in any position to make your program easier to read. The only restrictions are:

- a keyword must be preceded and followed by at least one blank.
- blanks are significant within string constants
- blanks are forbidden within numeric constants (including line numbers), keywords, variable and function names.

For example the statements:

```
20 INPUT "Length";L
```

and

```
20 INPUT        "Length";L
```

are equivalent but

```
20 INPUT "L e n g t h ";L
```

is a different statement as it contains a longer string constant.

3. CLASSIFICATION OF STATEMENTS ACCORDING TO FUNCTION

A BASIC program consists of a series of statements, that can be divided into two categories:

- Declarative Statements that:
  - . define the dimensions of an array (DIM)
  - . define a comman area (COMMON)
  - . define a user function (DEF FN)
  - . define an internal file (DATA).
- Executive Statements that describe the operations to be performed.

Statements may also be classified on a "user-oriented" basis, grouping together those statements which are used together in a program or which perform a similar function.
In this manual the latter method is used.
Chapters 1-7 describe the most important points of most of the BASIC statements. Refer to chapter 8 for a detailed and complete description of each single statement.

COMMENTS

You can insert a comment at any point in a program, in one of two ways:

- using the REM statement
- using comment fields (character strings preceded by an apostrohpe (')).

You can write any string of characters after the REM keyword.

For example:

```
100 REM BESSEL FUNCTIONS
.
.
.
350 'CALCULATION OF THE AREA OF A RECTANGLE
```

The statements DEFINT/DEFSNG/DEFDBL/DEFSTR can be used to explicitely declare the variables of a program, that is, to associate a type to each program variable.
These can be:

- numeric variables
- string variables
- numeric arrays
- a string array.

There are four types of simple data:

- integer
- single precision
- double precision
- string

and two types of subscripted data:

- array
- record.

An array is a series of simple variables (indexed variables) which are all of the same type.

A record is a series of simple or subscripted variables which may be of different types.

A type is associated to a constant when this appears in a BASIC statement according to the rules stated in chapter 4.

A type is associated to an expression when this is evaluated according to the rules in chapter 5.

A type is associated to a function value when this function is evaluated.

This type is also called the "function type". It depends on the algorithm used to calculate the function in the case of a user-defined function. Built-in numeric functions are of a pre-defined type.

A type is associated to a simple variable when the variable is implicitly or explicitly declared (see chapter 4).

A common type is associated to each element of an array when the array is implicitly or explicitly declared (see chapter 4).

The common type is also called "array type".

Conversion is automatic during assignments, argument-parameter transfer and during the reading of data from an internal or external file (See chapter 4).

See chapter 4 for rules on compatibility.

Simple numeric variables and elements of numeric arrays which have not been explicitly declared in a BASIC program, will be implicitly declared in single precision.

Array Dimensions

The dimensions of a numeric or string array and the upper and lower subscript bounds can be defined explicitly with the DIM statement and the OPTION BASE statement, otherwise they may be assumed implicitly.

ASSIGNMENT STATEMENTS

There are three assignment statements in BASIC:

- the CLEAR statement which sets all numeric variables to zero and initializes all string variables to null.
- the LET statement which assigns the value of an expression to a variable. The variables and the expressionsmust both be of the same type (either numeric or string).
- the SWAP statement which exchanges the values of variables provided they are of the same type (integer, single-precision, double-precision, string).

Numeric Assignments

If the value of the numeric expression is not the same as that of the receiving variable, BASIC converts the type of the expression value to the type of the receiving variable according to the rules described in chapter 5.
Rounding or overflow may occur, if the receiving variable is not able to contain the computed value.

String Assignments

String assignment is performed by moving the string expression value into the receiving variable, character by character, from left to right.

CONTROL STATEMENTS

Control statements alter the normal flow of program execution, by branching to another part of the program.
Branches may be conditional or unconditional.

Unconditional Branches

The GOTO statement causes an unconditional transfer of control. You simply indicate the number of the line to which control is to be transferred.

For example: 80 GOTO 20

Conditional Branches

You may use the following statements to branch to a specific statement:

- IF...THEN...ELSE
- IF...GOTO...ELSE
- ON...GOTO

Loops

You can create loops using:

- the FOR and NEXT statements, that are used to enclose a series of statements enabling you to repeat those statements a specified number of times
- WHILE and WEND statements that can be used to enclose a series of statements, enabling you to repeat these statements as long as a given condition occurs.

PROGRAM CHAINING

You may use the CHAIN and COMMON statements to chain a program.

The CHAIN statement with all its options provides you with a powerful and flexible tool for chaining a program

The CHAIN statement can be used in three distinct ways:

- in conjunction with one or more COMMON statements so as to pass common variables to the CHAINed program

- with the MERGE option in order to merge the CHAINed program with the program residing in memory (the DELETE option is often used in this case to delete a section of the program at the end of execution thus allowing overlays to be loaded in sequence).

- with the ALL option to pass all the variables to the CHAINed program.

FUNCTIONS AND SUBROUTINES

The main difference between a function and a subroutine is the way in which they are used. Both can use and modify the values of program variables, but a function is used mainly to compute a single value and return this to the statement that requested it.

Functions

A function is a series of pre-defined operations. BASIC will interpret the function when it finds its name in an expression.
A function computes a single numeric or string value depending on the type of expression used to define the function.
The name of the function may be followed by one or more "arguments" or by no argument. Arguments are separated by commas. They may be constants, variables or expressions. Parameters are also separated by commas, but a parameter may only be a variable.
The number of arguments must be the same as the number of parameters and their types (numeric or string) must match. The association between arguments and parameters is positional.

Each function can be called simply by stating its name followed in parentheses by one or more "arguments" representing the values to be passed to the parameters. We can classify BASIC functions into two main categories:

- built-in or system functions
- user-defined functions

Built-in or System Functions

Built-in functions are an instrinsic part of BASIC. They provide a set of commonly used numeric and string operations. They can be used in any program without them being explicitely defined. A complete list and a detailed description of system functions is provided in chap. 8.

Built-in Numeric Functions

BASIC provides a number of pre-written routines that permit certain mathematical functions to be calculated such as square roots, trigonometry, logarithms, etc. The results are integers or in double precision.

Built-in String Functions

These are instrinsic functions that calculate a string or a numeric value and permit one or more than one numeric and/or string argument(s) to be used. They simplify string operations such as the extraction of a group of characters i.e. a substring, from a larger string.

User-Defined Functions

The user can define an arbitrary number of functions within a BASIC program using the DEF FN statement. User-defined functions are called in exactly the same way as built-in functions. The only limitation is that the definition is program-dependent and must therefore be redefined in each program that needs to use it (unless the second program is chained to the first with the MERGE option).
DEF FN defines a numeric or string function.

For example:

```
10 DEF FNA(X)=(SIN(X/5)*3.1)/180
20 DEF FNB(X)+(FNA(X)+SIN(X))*.5
```

Subprograms

A BASIC subroutine consists of any sequence of BASIC statements that can be called repeatedly and it forms an integral part of the program.

A subroutine can be called by a GOSUB or an ON...GOSUB statement. At the end of execution of a subroutine, control is returned to the first statement following the most recent GOSUB (or ON...GOSUB) that has been executed.
A subroutine ends with the RETURN statement.
If a program refers to the same subroutine more than once, control is always returned from the subroutine to the statement following the most recent GOSUB (or ON...GOSUB) executed.

A subroutine may be called by another subroutine. In this case, the called subroutine is "nested" within the calling one. The number of "nested" subroutines that are active at this same time, is only limited by the amount of memory available.

BASIC provides a number of I/O statements. I/O statements that do not refer to external files will be dealt with first. The handling of external files will be described in the paragraph "EXTERNAL FILES".

Keyboard Entry

Two BASIC statements are provided for entry of data via keyboard during program execution. These are INPUT and LINE INPUT.
The INPUT statement allows you to enter one or more numeric or string data-items which will be assigned to the variable(s) specified in the statement. With LINE INPUT you can enter an entire input line and assign it to a string variable.

For example:

```
10 INPUT A,B
```

Program execution is suspended and a question mark (?) will be displayed. At this point the user is invited to enter the values of A and B. When all the requested values have been entered, execution restarts for the next executable statement.

Reading of Data from an Internal File

The DATA statements create an internal file i.e. a sequence of data (belonging to the program) which must be transferred to the program variables through the use of one or more READ statements.
The RESTORE statement repositions the pointer at the beginning of the file or at a specified line number.

Data Output

The PRINT or PRINT USING statements are used to display data on a screen. The PRINT statement displays the results in standard format while PRINT USING allows a user-defined format which is specified by the string expression which appears after USING.

This expression may be a constant or a string variable containing special formatting characters. These characters determine the fields and the format of the output strings or numbers. The Output of data can be directed to a printer through the use of the statements LPRINT or LPRINT USING.

EXTERNAL FILES

External files are a collection of data, that are external to a BASIC program. BASIC allows the user to create, open, use, close and delete files. Files can also be locked.

The main characteristics of external files are illustrated below:

	ACCESS MODE		
File Organization	Sequential	Direct by address	Direct by key
SEQUENTIAL	YES	-	-
RANDOM	YES	YES	-
KEYED	YES	-	YES

Tab. 3. 1 - File Organization and Access Modes

File Organization The way in which a file is organized establishes the way in which it can be accessed. File organization may be established by using:

- SHELL commands or the CREATE statement
- I/O statements associated with the file.

Sequential

A sequential file is simply a sequence of data-items without any grouping criterion.
The number of data-items involved in an Input/Output operation can vary and it is specified through the use of READ and WRITE statements. Sequential files allow:

- the reading of a file from the beginning to the end (one data-item after the other) i.e. it is not possible to re-read an item that has already been read.

- the sequential writing from the beginning of the file, with the possibility of appending new data-items at the end of the file.

Random

A random file is a series of data-items grouped in records. As a result each I/O operation involves one record at a time.

The length of each individual record (which is established through the CREATE statement) is a fixed length, even though this may vary from file to file. Each record of the file can be accessed either directly, on the basis of the record number associated to each record or sequentially, in ascending numerical order. This type of file allows any record to be read or written independent of all the others.

Keyed

A keyed file is a series of data-items grouped together in records. Each I/O operation involves one record at a time.
There is a fixed length for each individual record even though this may vary from file to file. Files are organized on the basis of keys (single or multiple). Records are accessed either directly using their key or sequentially according to ascending key order starting from any point of the file.
A keyed file allows any record of the file to be read, written, re-written or deleted, independent of all the others.

Creation of a Disk File

Before opening an external file, which is on disk, it must be created with the CREATE statement. This statement assigns a name to the file and establishes the file organization.

The Opening of a File

The OPEN statement opens a disk file allowing a program to perform Input/Output operations on the file.
The OPEN statement performs various functions:

- it defines the access mode which may be:
  - . INPUT: prepares the squential reading of data starting from the beginning of a file
  - . OUTPUT: prepares the sequential writing of data starting from the beginning of the file
  - . APPEND: prepares the reading of data starting from the end of the file
  - . RANDOM: records can be read/written in any order
  - . KEYED: records can be read, written or deleted in any order
- provides file protection, if required. The LOCK clause prevents any other user from accessing the file until the file has been CLOSEd.
- determines record length for random access or keyed files.
- gives a file name

  The file name may indicate:
  - . a complete pathname
  - . the name of the file in the work directory
  - . a local name which is assigned a global value, only before BASIC is started under SHELL
- associates a file indicator and a physical file. Once they have been associated the file indicator can be used in any I/O operation to identify the physical file until a CLOSE statement is executed on the same indicator.

The file indicator is expressed in the OPEN statement from a numeric expression whose value (rounded to the nearest whole number) is from 1 to 15.

The Closing of a File

The CLOSE statement allows the user to close one or more files by simply writing one or more file indicators after the keyword CLOSE.
Once a file has been closed, it can be re-opened by using the same or another file indicator.

For example:

```
100 CLOSE 1,2,3
```

The END statement closes all open files.

Deletion of a Record

A record can be deleted from a keyed file with the statement

```
REMOVE #
```

The record identified by the value of the primary key is removed from the file.

Input/Output Statements

I/O statements vary according to file organization.

1. Sequential files.

The input statements are:

- INPUT #
- LINE INPUT #

The output statements are:

- PRINT #
- PRINT # USING
- WRITE #

Access is sequential.

2. Random and Keyed files,

The input statements are:

- GET #

The output statements are:

- PUT #

SUSPENSION OF PROGRAM EXECUTION

Program execution is suspended when:

- /CTRL/ /C/ is entered, or
- a STOP statement is executed or
- an error message is issued.

The System will enter Command Mode in the above cases. (In the case of a syntax error, however, Editor Mode is entered).
In Command Mode, the user can display the program variables (using the immediate statements PRINT and PRINT USING) or modify the value (using the LET or CLEAR statements).
It is possible to resume execution by entering a CONT command. Program execution cannot be resumed if it has been modified or if an error has been detected.

4. DATA

Each data-item may appear in a BASIC program as either a constant or a variable.

Constants may be:

- numeric
- string.

Variables may be:

- numeric
- string.

Variables may also be simple or subscripted (arrays).

CONSTANTS AND VARIABLES

Constants

Constants are specific numbers such as 15, -2, 3.41 etc. or specific strings such as "AAA.b1", "Cursor***". This means that their values remain the same throughout program execution.

Variables

A variable is a named data-item whose value may change during program execution.

Variable Names

The identifier (or name) of a variable may be of any length, but only the first 40 characters are significant. The characters allowed may be letters or numbers. The period (.) is also allowed. The first character must be a letter. The last character may be a letter, a number, a period, or a type declaration tag (%,!, # ,$). The meaning of a type declaration tag is explained later in this chapter. Lower-case letters in a variable identifier are considered equivalent to their corresponding upper case letters and are converted to their corresponding upper case letters when the program is listed. A reserved word (a keyword, a command or function name) cannot be used as a variable identifier, but BASIC allows reserved words to be embedded within, before or after a variable identifier.

For example:

```
10 PERFORMANCE = 105.3
20 SINGLE = 1371.2
```

are valid program lines even though PERFORMANCE contains the keyword FOR and SINGLE begins with the name of the built-in function SIN.

HOW BASIC CLASSIFIES CONSTANTS

The way BASIC stores a data-item determines:

- the amount of memory (in bytes) that it will occupy
- the speed at which BASIC can process it.

Numeric Data

BASIC can store all numbers in a program as follows:

- integers (fastest processing, speed limited range)
- single precision numbers (general purpose), or
- double precision numbers (maximum precision, slower processing speed).

	INTEGER NUMBERS	SINGLE PRECISION NUMBERS	DOUBLE PRECISION NUMBERS
Memory space (in bytes)	2	4	8
Range of values	-32768 to 32767	$\pm 10^{38}$ to $\pm 10^{-38}$	From $\pm 10^{-308}$ to $\pm 10^{308}$
Significant Digits	Up to 5	Up to 7	Up to 16
Displayed Digits (PRINT/LPRINT)	Up to 5	Up to 6 (with rounding)	Up to 15 (with rounding)

Tab. 4. 1 - Numeric Data

String Data

Strings (sequences of ASCII characters) are useful for storing non numeric information
For example, the constant:

"FORD, RENAULT"

is a quoted string of 13 characters. Each character in the string (including the blank) is stored in one byte and corresponds to an ASCII code.

A string can be up to 255 characters long. A string with length zero is called a "null" string and is represented by a pair of double quotes (""). BASIC allocates strings dynamically i.e. the memory space reserved for a string may vary during program execution from 0 to 255 bytes.

NORMAL CRITERIA TO CLASSIFY CONSTANTS

IF....	THEN....	EXAMPLES
the value is enclosed in double quotes	it is a string	"NO" "YES" "Circle" "" (null string)
the value is not in quotes	it is a number an exception to this rule is during data input, in reply to an INPUT or LINE INPUT statement and in DATA statements.	521 - 15 3.7345E-2 43#
a number is whole and in the range -32768 to 32767	it is an integer constant	1024 721 -32768
the value has the prefix &H, and is enclosed between the digits 0-9 and the letters A-F (in the range 0 to FFFF)	it is a hexadecimal constant.	&H20F0 &HF1 &H35 &HFE98 &HFFFF &H0
the value has the prefix &0 or & and is enclosed between the digits 0-7 (in the range 0 to 177777)	it is an octal constant	&070 &044 &71175
a number is not an integer and does not contain more than 7 digits	it is a single precision constant	-2.3 32768 45.314 -65000
a number contains more than 7 digits	it is a double precision constant	52174593 -54.397124 8.799999999

TYPE DECLARATION TAGS

The user can override BASIC's normal criteria by adding the following "tags" to the end of a numeric constant:

TAG	MEANING	EXAMPLES
!	declares a number in single precision	5.72110333! the constant is in single precision and only the first 7 digits are memorized (i.e. 5.721103)
E	single precision floating point. The E indicates that the constant is to be multiplied by the power of 10 specified after E	7.31E4 means 7.31×10^4 i.e. 73100
#	declares a number in double precision	4# 5.21#
D	double precision floating point. The D indicates that the constant is to be multiplied by a power of 10 specified after D.	7.2D-3 means 7.2×10^{-3} i.e. 0.0072

HOW BASIC CLASSIFIES VARIABLES

Each variable identifier that appears in a BASIC program, can be classified as either a string, or integer or as a single or double precision number.

Unless otherwise indicated, BASIC classifies all numeric variables in single precision.

However the user may assign different type attributes to variables using either definition statements or a type declaration tag at the end of the variable identifier.

BASIC provides four type definition statements (DEFINT/DEFSNG/DEFDBL/DEFSTR).
A type definition statement declares the type of all the variables whose names begin with a specified letter.

Type Declaration Tags

As with constants, the user can override the type of a variable by adding a type definition tag at the end of the name. There are four type declaration tags for variables.

TAG	MEANING	EXAMPLES
%	integer variable	A% STEP% INCREMENT% are all integer variables, regardless of the type definition statements for the variables beginning with the letters A, S and I.
!	single precision variable	SPEED! SPACE! TIME! are all single precision variables regardless of the type definition statements for the variables beginning with the letters S and T
#	double precision variables	TOTAL # SUBTOTAL# X1 # are all double precision variables regardless of type definition statements for the variables beginning with the letters T,S and X
$	string variable	A$ B1$ NAME CLASS$ are all string variables regardless of the type definition statement for the variables beginning with the letters A, B and N.

Often a program or immediate line may ask BASIC to assign one type of constant to a different type of variable. At this point the conversion procedures described below are used.

Single or Double Precision to Integer

BASIC converts the original value in single or double precision to an integer by rounding the fractional part.
The fractional part must be greater than or equal to -32768 and less than 32767 otherwise an Overflow occurs.

For example:

```
C%=-15.1
Ok
?C%
-15
Ok
```

the value -15 is assigned to the variable C%.
For example:

```
C% = 4.1E2
Ok
?C%
 410
Ok
```

the value 410 is assigned to the variable C%
For example:

```
C% = 47.8
Ok
?C%
 48
Ok
```

the value 48 is assigned to the variable C%

an overflow error occurs.

Integer to Single or Double Precision

In this case the conversion does not cause an error. The converted value corresponds to the original value with zeros to the right of the decimal point.

For example:

```
S!= 326
Ok
?S!
 326
Ok
```

326 is stored in the variable S! as 326.000 but it is displayed as 326

For example:

```
D# = 326
Ok
?D#
 326
Ok
```

326 is stored in D#as 320.0000000000000 but displayed as 326.

From Single to Double Precision

BASIC adds trailing zeros to the single precision number.
If the original value:

- has a binary representation, no error occurs in the conversion
- does not have an exact binary representation, an arithmetic error is introduced when converting the value.

For example:

```
B# = 1.5
Ok
?B#
 1.5
Ok
```

When entering B# = 1.5 the value 1.500000000000000 is assigned to variable B# but only 1.5 is displayed.

1.5 has an exact binary representation.

For example:

```
C#=1.3
```

```
Ok
?C#
 1.29999995231628
Ok
```

When entering C# = 1.3 the value 1.299999952316280 is assigned to variable C# but it is only displayed as 1.29999995231628.

Double to Single Precision

This involves converting a number with up to 16 significant digits into a number that has no more than 7.
Only the first seven digits of the original value are valid and the last digit is rounded. Before displaying or printing such a number, BASIC rounds it down to six digits.
If the double precision value is outside the range of the single precision values, an Overflow occurs.

For Example:

```
P!= 2.03999996
Ok
?P!
 2.04
Ok
```

the value 2.040000 is assigned to the variable P! but it is only displayed as 2.04

Illegal Conversions

The user cannot convert numeric values to string values and vice versa by means of an assignment statement.

For example:

```
C$=321.7
```

is illegal. (In these cases, conversion takes place with the functions STR$ and VAL (see Chap. 8)).

SUBSCRIPTED VARIABLES AND ARRAYS

An array is a collection of variables of the same type under the same name, which can be distinguished by the value of one or more subscripts.
In BASIC an array can have a maximum dimension number of 10. To define an array, the user must:

- give it a name (the rules given for naming simple variables apply)

- establish the upper and, lower subscript bounds.

To do this, the user must write a DIM statement and if necessary, an OPTION BASE statement. If the following is specified in a program:

```
10 OPTION BASE 1
```

the lower bound of all arrays is 1.
If no OPTION BASE statement is present in the program (or if there is the statement OPTION BASE 0) the lower bound of all arrays is 0.
It is also possible to redefine an array by means of a new DIM statement that is preceded by an ERASE statement.

5. EXPRESSIONS

BASIC can process three types of expressions:

- numeric expressions
  For example:

  A+B*C

- string expressions
  For example:

  A$+B$

- logical expressions
  For example:

  A+B>C

The value of a numeric expression is a numeric value (for example 350.71). The value of a string expression is a string value (for example "ABC"). The value of a logical expression is a boolean value (true or false).

NUMERIC EXPRESSIONS

A numeric expression contains one or more operands (constants, simple variables, array elements, or functions) which are connected by means of numeric operators. (+,-,*,/,\,MOD,-,^).

Numeric Operators

BASIC uses operators to define numeric operations. There are 8 numeric operations, each identified by its own symbol.

SYMBOL	OPERATION	EXAMPLES
+	addition	X = 3.2 Ok ?X+1.1 4.3 Ok
-	subtraction	?X-1.3 1.9 Ok
\	integer division. The operands are rounded to the nearest integers (which must be in the range - 32768 to 32767) before the division is performed and the quotient is truncated to an integer.	?10\4 2 Ok ? 25.68\6.99 3 Ok
MOD	modulus arithmetic. Provides the integer value which is the remainder of an integer division	?10.4 MOD 4 2 Ok (10/4 = 2 with remainder 2) ?25.68 MOD 6.99 5 Ok (26/7 = 3 with remainder 5)
*	multiplication	?X*3.92 12.544 Ok
/	division	?3/6.05 0.495868 Ok
-	negation	?-X -3.2 Ok
^	exponentiation	?X^3 32.768 Ok

Tab. 5. 1 - Numeric Operators

Numeric Operator Priority

BASIC has priority levels for performing different numeric operations, when these are present in the same expression.
For operators with the same priority (e.g. / and *) operations are carried out from left to right.

Numeric operators and priority levels are described below (in order of descending priority).

ORDER OF PRIORITY

1. exponentiation
2. negation
3. multiplication and division
4. integer division
5. modulus arithmetic
6. addition and subtraction.

Use of Parenthesis

There are cases when the normal priority of operations must be changed. To do this you use pairs of parentheses exactly as you would in algebra. When parentheses are used, the operations within the innermost pair of parentheses are performed first, followed by operations within the second innermost pair and so forth. Within a given pair of parentheses, normal priority of operations applies.

NUMERIC EXPRESSION	BASIC EQUIVALENT	INTERPRETATION
$x+y+z/2$	(X+Y+Z)/2	1. Add X,Y and Z 2. Divide the sum by 2
$x+\frac{y+z}{2}$	X+(Y+Z)/2	1. Add Y to Z 2. Divide the sum by 2 3. Add X to the result
$2x + 5$	2*X+5	1. Multiply X by 2 2. Add 5 to the result
$2(x+4)$	2*(X+4)	1. Add 4 to X 2. Multiply the sum by 2
$x^2 + 3$	X^2+3	1. Square X 2. Add 3 to the result
$(x + 3)^2$	(X + 3)^2	1. Add 3 to X 2. Square the result
$\frac{(x + 3)^2}{4}$	(X+3)^2/4	1. Add 3 to X 2. Square the sum 3. Divide the result by 4
$\frac{x^2}{6} \cdot \frac{x + y}{2}$	(X^2/6)*(X+Y)/2))	1. Square X and divide the result by 6 2. Add X to Y and divide by 2 3. Multiply the two results by each other.

Tab. 5. 2 - Examples of Numeric Expressions

Type of Expression

The type of numeric expression depends on the type of its operands.
If the expression involves more than 2 operands, it can be considered as a series of calculations involving two operands.

There are four possible situations depending on the

type of the two operands involved:

- if both operands are of the same numeric type (integer, single precision or double precision) the result is also of that type.

- if one operand is integer and the other is single precision: the result is single precision

- if one operand is integer and the other is double precision: the result is double precision

- if one operand is single precision and the other is double precision: the result is double precision.

Rounding, Overflow and Underflow

Floating point types are forms of approximation to the real numbers of mathematics.

- if one or more operands in a numeric expression are floating point, calculations are approximate and accuracy can be lost.
  If this occurres the less significant digits are not regarded and the last digit maintained is rounded.

- if the value of the expression exceeds the maximum value allowed for that data-type, an "Overflow" message is displayed; machine infinity with the algebraically correct sign is supplied as the result and execution continues.

- if a division by zero is enconutered out, the "Division by zero" error message is displayed; machine infinity with the sign of the numerator is supplied as the result of the division and execution continues.

- if the evaluation of an exponentiation results in zero being raised to a negative power, the "Division by Zero" error message is displayed; positive machine infinity is supplied as the result of the exponentiation and execution continues.

- if the value of the expression is less than the smallest representable value; the value becomes zero (Underflow) and execution continues.

- if in a numeric assignment the execution type is different from the type of the receiving variable, the expression type is automatically converted to the type of receiving variable.

Note — Machine infinity is displayed as 1.79769313486231 D +308

Undefined Values — If a numeric variable in a numeric expression has not yet been initialized, it is set to the value zero.

Undetermined Forms — The evaluation of a numeric expression may result in an undetermined form, such as:

0/0: the message "Division by Zero" is displayed and the value 1.79769313486231 D +308 (machine infinity) is supplied

0^0 : the value equal to 1 is assumed

STRING EXPRESSIONS

A string expression can be either a string constant, a single string variable, a string array element, a string function or a chaining of all these elements through the use of the plus sign (+).

Concatentation of Strings

Strings can be joined i.e. "concatenated"; The result of this operation is the string obtained by joining the start of the second string to the end of the first.
When two or more strings are concatenated, the length of the resulting string is equal to the sum of the individual string lengths.

Null Strings

The operand in a string expression may also be a null string (""). The null string may also be the default value of a non-initialized string variable.

Note

Care must be taken not to assign more than 255 characters to a string variable, otherwise the system issues the message "String too long".

RELATIONAL EXPRESSIONS

Relational expressions compare either two numeric or two string expressions by means of a relational operator.

Relational operators

The relational operators are:

= equals

> greater than

< less than

> = or = > greater than or equal to

< = or = < less than or equal to

<> or >< not equal to

It is illegal to compare a numeric expression with a string expression and vice versa. Comparison of numbers has an obvious meaning. Character strings may also be compared depending on the numeric value of the character's representation. (This is established on the basis of the ASCII decimal value).
String scanning is performed from left to right. Numeric or string expressions are performed first, then relational operators are applied to the result of such expressions.

For example, to write:

A>B + C

is equivalent to

A > (B + C)

The result of the relational expression is numeric. It is displayed as either +1 (if the relation is true) or 0 (if it is false).

Using Relational Expressions

The result of a relational expression may be used to make a decision regarding program flow. It is possible to use relational expressions in the following control statements:

- IF... GOTO... ELSE, or
- IF... THEN... ELSE, or
- WHILE

where a condition is tested to determine subsequent operations in the program.

For example, the following statement:

```
100 IF A$>B$ THEN 50
```

will transfer control of execution to statement 50 if the condition (A$>B$) is true. If the condition is false, (i.e. A$ is not greater than B) the next statement will be executed.

LOGICAL EXPRESSIONS

A logical expression consists of one operand preceded by the logical operator NOT, or two operands separated by another logical operator (AND, OR, XOR, EQV and IMP) or two operands separated by a logical operator and NOT.

The operands in a logical expression may be numeric or relational expressions. Both provide a numeric result. The result of a logical expression is also numeric. Examples (of logical expressions) are:

```
NOT X
X AND Y
A>B OR C>D
I% AND A$<B$
A$ XOR B$ not valid (as the operands are strings).
```

Logical Operators

Logical operators work by converting their operands to sixteen bit, signed two's complement integers in the range -32768 to +32767. If the operands are not in this range, an error occurs. If both operands are 0 or -1 the logical operators will provide the result 0 or -1.
The given operation is performed on these integers, where each bit of the result is determined by the corresponding bits in the two operands.
The logical operators are listed below in a 'truth table'.
This describes graphically the results of the logical operations on a bit-by-bit basis.

A	NOT A	A	B	A AND B	A OR B	A XOR B	A EQV B	A IMP B
1	0	1	1	1	1	0	1	1
0	1	1	0	0	1	1	0	0
		0	1	0	1	1	0	1
		0	0	0	0	0	1	1

Tab. 5. 3 - Truth Table

Logical Operator Priority

In an expression, logical operations are performed after numeric and relational operations.

The table below lists logical operators. The operations are given in descending order of priority.

PRIORITY

1. NOT
2. AND
3. OR
4. XOR
5. IMP
6. EQV

Using Logical Expressions

You can use logical expressions:

- to test a condition in the following control statements:

  IF... GOTO... ELSE
  IF... THEN... ELSE

  For example, the following statement:

  50 IF A$>B$ and B<=C THEN 300

  will transfer control of execution to statement 300 if the condition A$>B$ and B<=C is true. If the condition is false, the next statement will be executed.

- to test words (16 bits) for a particular bit configuration.

For example, the AND operator may be used to "mask" all but one of the bits of a status word associated to an I/O buffer.
The OR operator can be used to create a particular binary value.

For example:
1 AND 8 is 8
1 OR 8 is -1

as -1 binary = 111111111111111
8 = binary 0000000000000000

OPERATOR PRIORITY

The table below lists all the operators (numeric, string, relational, and logical) in the order in which BASIC evaluates them.

1. ^ (exponentiation)
2. - (negation)
3. */ (multiplication and division)
4. \ (integer division)
5. MOD (Modulus arithmetic)
6. + - (addition and subtraction)
7. + (chaining of strings)
8. All relational operators
9. NOT
10. AND
11. OR
12. XOR
13. IMP
14. EQV

Operators shown on the same line have the same priority

All relational operators have equal precedence.

Evaluation order of expressions can be overridden by use of parentheses.

For example, the evaluation order of:

```
NOT A>B AND C>D OR E>F
```

is different from the evaluation order of:

```
NOT (A>B  AND (C>D OR E>F))
```

6. DEBUGGING AND ERROR RECOVERY

Leaving aside errors made when entering a line, there are two types of error that can be made:

- run-time errors which halt execution and which cause an error message to be displayed

- logic errors which permit complete program execution but cause incorrect or unexpected results.

Run-time errors may be Syntax errors or other types of run-time errors (NEXT without FOR, RETURN, without GOSUB, etc.).
You can also simulate a BASIC error or generate a user-defined error (to be handled by an error trap routine). See ERROR and ON ERROR GOTO statements in Part II.

Tracing Program Execution

A convenient method of debugging logic errors is to trace the order of statement execution in all or part of a program.
BASIC provides the following two tracing commands (TRON/TROFF) that may also be used as program statements and which cause the line number of each statement executed to be listed (TRON) and stop the line number listing (TROFF).

Error Testing and Recovery

Normally, when a run-time error is encountered, BASIC handles the error by halting execution, displaying an appropriate message and returning to Command Mode.
In some cases the user may want to handle the error in a different way. This can be done by writing an error-handling routine.

Through the use of the ON ERROR GOTO statement, it is possible to enter error-handling routines. Control is passed to the line specified after GOTO when an error occurs.

Only one ON ERROR GOTO statement can be used in a program for non standard error-handling.
Execution of an ON ERROR GOTO 0 outside an error handling routine, disables error trapping. Execution of an ON ERROR GOTO 0 inside an error handling

routine specifies normal error handling for any error that the routine does not handle.

In the case of a run-time error where error trapping has been enabled, control of execution is passed to the specified line. Then the ERR and ERL functions can be used to execute the error recovery routines. The ERR function contains the error code and the ERL function contains the line number where the error has been detected.
The error handling routine tests all the particular errors that the user wishes to recover and it indicates the way in which to handle them. This usually involves correction of the error and execution is resumed at the statement where the error occurred, without returning to Command Mode.

7. MULTIPLE WINDOWING

The screen can be subdivided into rectangular areas called "windows".
A maximum of 16 windows can be opened at the same time and their dimensions are established when the user enters the WINDOW(Def) function. A window can be used in exactly the same way as the entire screen. Operations carried out on different windows are completely independent.
The origin (the position in which the first character will be displayed) is the upper left-hand corner of window. However, you can move to any character position within the window by using the CURSOR statement.

The co-ordinates of each text character are expressed in terms of column and row and always in relation to the entire window.

Each window has a text cursor which you can position anywhere on the screen (using the CURSOR statement).

The text cursor automatically moves one position each time a character is entered.

Opening Windows

Once in the BASIC environment the entire screen is a single window.
You may define a new window by simply dividing the existing window vertically or horizontally (WINDOW (Def) function).

When you open a new window, the previous contents of that area are cleared.

You can use the FRAME statement to frame each new window.

Using Windows

You can operate on any of the windows (WINDOW(Sel) statement). You can clear the contents of a window with the CLS statement.

Closing Windows

You can close any of the windows at any moment. You can also return to the "initial system" state where there is only one window (the whole screen).
To perform either of these operations, the CLOSE WINDOW statement is used. The area of the window that is closed, is assigned to the nearby windows.

8. BASIC STATEMENTS, COMMANDS AND FUNCTIONS

This chapter contains a detailed description of all the M30/M40 BASIC statements, commands and functions. They are listed in alphabetical order for easy reference.

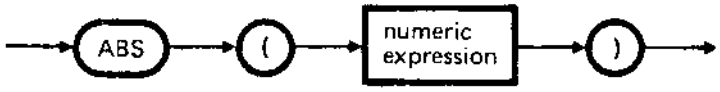
Description of Statements, Commands and Functions

The description of BASIC statements,commands and functions is ordered as follows:

Function:	describes the function of the statement, command or function.
Format:	describes the syntax
Characteristics and Use:	describes in detail how the system operates when executing a statement command or function together with other considerations.

ABS (ABSOLUTE VALUE)

This function calculates the absolute value of a numeric expression.

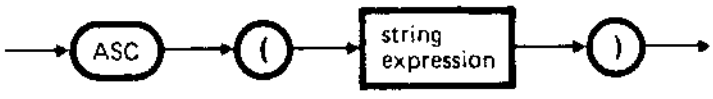


Example

```
PRINT ABS(7*(-5))
 35
OK
```

ASC (ASCII)

This function calculates a numeric value, which is the ASCII decimal code of the first character of a given string.



Note

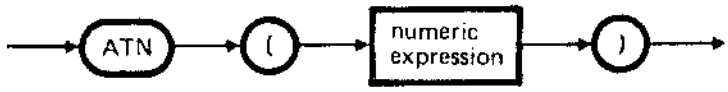
See the CHR$ function for converting from ASCII decimal code to characters.

Example

```
10 X$ = "TEST"
20 PRINT ASC(X$)
RUN
 84
OK
```

ATN (ARC-TANGENT)

This function provides the arctangent of the argument.



Characteristics

The result is expressed in radians within the range $-\pi/2$ and $+\pi/2$. Calculation of ATN is carried out in double precision.

Note

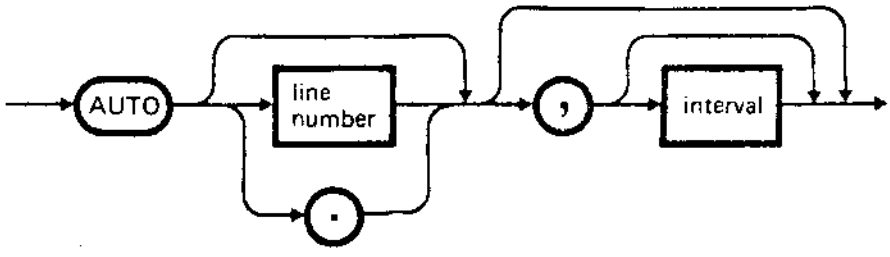
The numeric expression can be of any type; the result provided is in double precision. It is advisable to use a double precision numeric expression as input.

Example

```
10 INPUT X
20 PRINT ATN(X)
OK
RUN
? 3
1.24904577239825
OK
```

Automatically numbers the program lines.

IMMEDIATE Command.



where:

line number	is the first line number generated. The default value is 10.
.	the first number of the generated line is the number of the current line (the last one memorised).
interval	is the interval between line numbers. The default value is 10. If the comma is inserted and the interval is omitted, the interval value is the last increment specified.

Characteristics

If the AUTO command generates a line number which already exists, after the line number an asterisk will appear on the screen warning the user that a line previously memorised is about to be replaced. By pressing the carriage return/line feed key, the existing line will remain unchanged and another line number will be generated.
To terminate automatic numbering, the user must

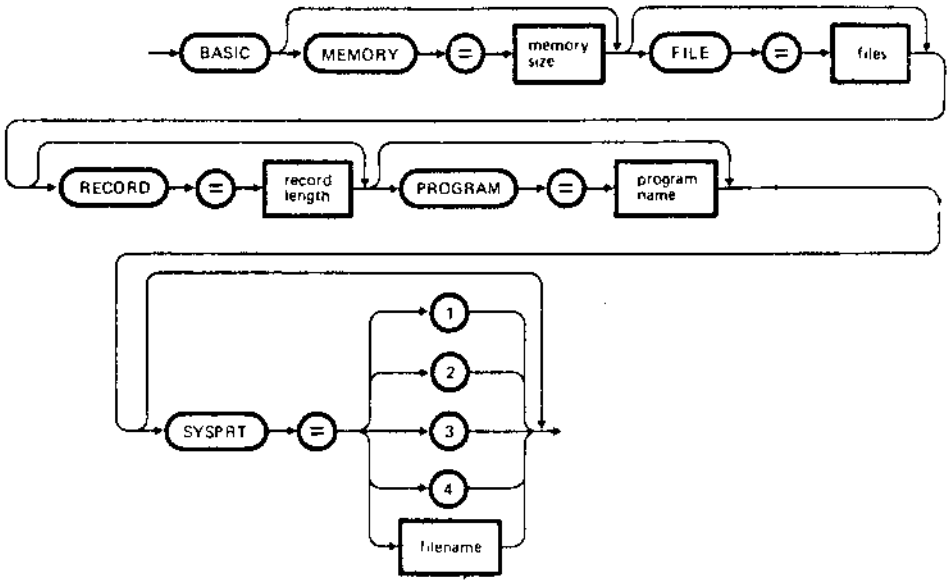
enter the control character /CRTL/ /C/; in this way, the line currently being operated on is deleted and the system returns to the Command State.

Examples

```
AUTO

AUTO .,30

AUTO 100

AUTO 150

AUTO 200,20
```

Sets the global parameters for BASIC programming.

IMMEDIATE Command



where:

memory size — is the maximum number of bytes available for the user memory in the BASIC environment. The permitted values are between 0 and 64k.
The default value is 64k.

files — is the maximum number of files which can be opened concurrently in BASIC. The permitted values are between 1 and 15.
The default value is 15.

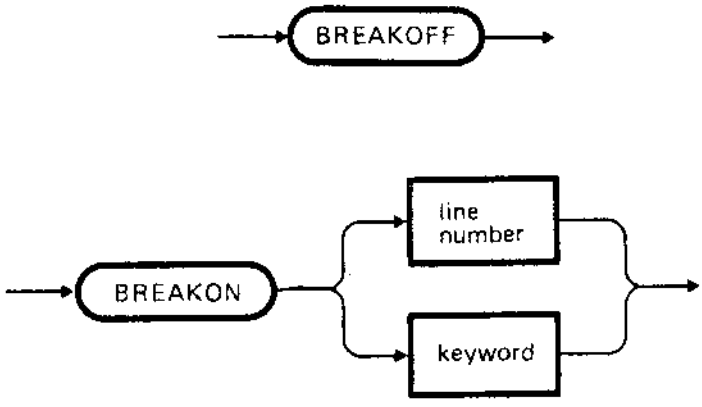
record length	is the maximum length of a record. The permitted values are between 1 and 4096. The default value is 256.
program name	is the name of the program to be executed. At end of execution, the system returns automatically to the SHELL environment.
sysprint	selects the device on which output is directed. If omitted, the output is directed to the work-station printer (if connected); otherwise to the standard output device (the screen). If a value 1,2,3 or 4 is associated to sysprint, output will be directed to the system printer selected by this value. Output may also be directed to a file by simply assigning the file name to sysprint.
Note	The parameters that activate the BASIC environment are of a "keyword type", and may thus be listed in any order.

BREAKOFF/BREAKON

BREAKON interrupts program execution at the specified line number or keyword.

BREAKOFF disables the use of the BREAKON command.

IMMEDIATE Commands.



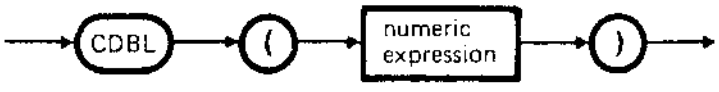
Characteristics

Two BREAKON commands can be active at the same time: one for the line number and one for the keyword.

After execution of the BREAKOFF command, program execution is resumed by entering the CONT command.

CDBL (CONVERT TO DOUBLE PRECISION)

This function converts any numeric format to an argument in double precision.



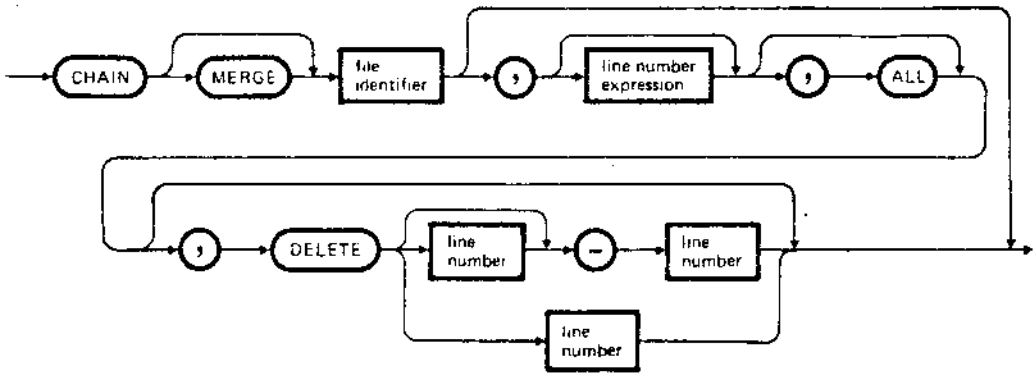
Example

```
10 A = 454.67
20 PRINT A; CDBL(A)
RUN
 454.67      454.670013427734
OK
```

CHAIN

Allows the specified program to be chained to the one in memory thus enabling variables to be transferred.

PROGRAM Statement



where:

MERGE — if the CHAIN statement includes the MERGE option, a MERGE operation is executed between the program in memory and the chained program.
The latter must be stored in ASCII format.
If the MERGE option is omitted, the program in memory is deleted and replaced by the chained program (excluding "common" variables).
MERGE is frequently used with the DELETE option and the line number expression to carry out a sequence of "overlays".
The MERGE option changes the variables type if the type definition statements are not repeated in the called program. When using the MERGE option, the variables that are initialised in the calling program are reinitialised if the COMMON statement or

the ALL option has not been used in the calling program.
The CHAIN statement with the MERGE option leaves the data files open and the previously entered OPTION BASE unchanged.

file identifier — is a string expression specifying the program to be chained.

line number expression — is a line number of the chained program or an expression whose value represents a line number. This number indicates the line to be executed first. If it is omitted, the chained program is executed from the beginning. Use of the RENUM command does not alter the value of the line number expression in any way.

ALL — if the CHAIN statement includes the ALL option, the values of all the program variables in memory are transferred to the chained program.
If the ALL option is not specified, the program in memory must contain a COMMON statement to allow the transfer of data.

DELETE — specifies (by means of a pair of line numbers) that a portion of the program in memory must be deleted. Deletion takes place before the chained program is loaded in memory.
DELETE is frequently used together with MERGE and line number expression to carry out a sequence of "overlays".
Line numbers used after DELETE are altered if the RENUM command is executed.

Characteristics — Possible type-definition statements (DEFINT, DEFSNG, DEFDBL, DEFSTR) relating to "common" variables, must also appear in the chained program (i.e. so that variable types are uniform).

Example 1

```
10 REM PROG1
20 COMMON A1,B1,C1$
 .
 .
 .
100 CHAIN "PROG2"
110 END
```

The program PROG1 chains program PROG2 and transfers the A1,B1,C1$ values to PROG2.

Example 2

```
10 REM PROG2
20 COMMON A2$,B2$
 .
 .
 .
80 CHAIN "PROG3",200
90 END
```

The program PROG2 chains PROG3 and transfers the values of A2$ and B2$.
Program PROG3 is executed starting from line 200.

Example 3

```
10 REM PROG 10
 .
 .
 .
50 CHAIN "PROG11", 100, ALL
60 END
```

The program PROG10 chains PROG11 and transfers all its variables to it.
PROG11 is executed starting from line 100.

Example 4

```
10 REM ROOT
 .
 .
 .
100 CHAIN MERGE "OVERLAY1", 1000
110 END
```

Using the MERGE option the ROOT program chains the OVERLAY1 program (recorded in ASCII format).
The latter is executed starting from line 1000.

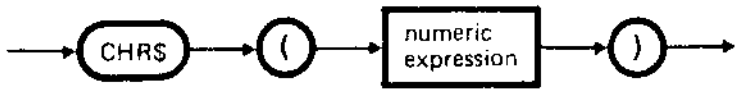
Example 5

```
1000 REM OVERLAY1
 .
 .
 .
1500 CHAIN MERGE "OVERLAY2",
     1000, DELETE 1000-1500
1510 END
```

Using the MERGE option the program OVERLAY1 chains the program OVERLAY2 (recorded in ASCII format).
The latter is executed starting from line 1000.
Before it is loaded in memory, the contents from line 1000 to 1510 are deleted.

CHR$ (CHARACTER)

This function provides the character whose ASCII decimal code is the value of the argument.



where:

numeric expression — is rounded up to the nearest integer. It must be within the range of 0 to 255 and it is interpreted as an ASCII decimal code.

Characteristics — The function CHR$ is frequently used to send a special character to the terminal or to the printer. For example, the character BEL (CHR$(7)) can be given as a preface to an error message.

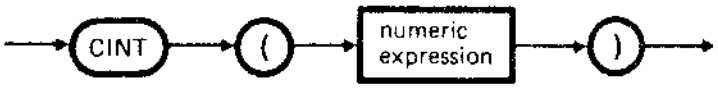
Note — See Appendix C for all the special characters or sequence of characters that can be transmitted to the printer or to the terminal via the CHR$ function.

Example

```
PRINT CHR$(66)
B
OK
```

CINT (CONVERT NUMERIC TO INTEGER)

This function converts any numeric argument into an integer, by rounding its fractional part (if the fractional part is $\geq$ .5 it is rounded up, if not it is rounded down).



Characteristics

The argument of the function CINT must be included within the values -32768 and 32767.

Note

See the functions CDBL and CSNG for converting numbers into single and double precision formats and the functions FIX and INT for converting into integer format.

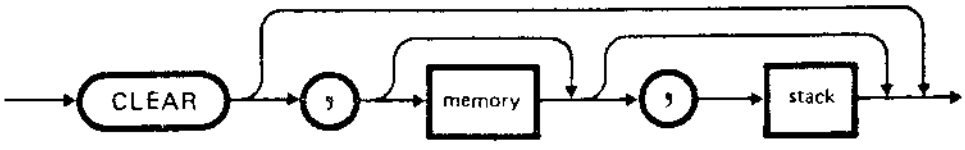
Example

```
PRINT CINT(45.67)
 46
OK
PRINT CINT(45.45)
 45
OK
```

CLEAR

Makes the numeric variables zero and initialises the string variables to the string value null.
It also establishes the memory space dedicated to the stack.
Moreover, it makes the entire screen zero, closes all the windows and returns to its initial state with only window 1 active.

PROGRAM/IMMEDIATE Statement



where:

memory — this parameter in the CLEAR statement has no effect at all: it is accepted for compatibility. The value of available memory is that assigned when BASIC environment was started, by means of parameter memory (see the BASIC command). It cannot be changed in the BASIC environment.

stack — establishes the memory space dedicated to the stack. The default value is the smallest value between 512 bytes and 1/8 of available memory space.

Characteristics — At the start of the BASIC program execution, even without the use of the command CLEAR, all numeric variables are made zero and all string variables are initialised to string value null (except for variables defined as "common" in CHAINing one

program to another).

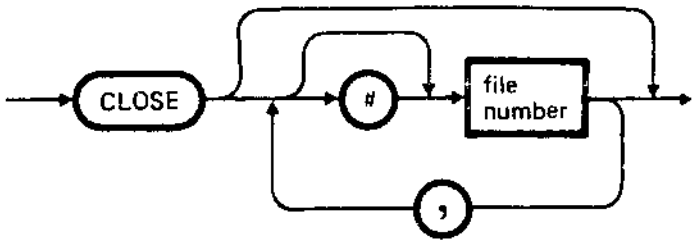
Example

```
CLEAR
CLEAR ,32768
CLEAR ,,2000
CLEAR ,32768,2000
```

CLOSE

Closes I/O operations on a data file on disk.

PROGRAM/IMMEDIATE Statement



where:

file number — is a numeric expression. This rounded value represents the number of the file, assigned with the corresponding OPEN. This number must come between 1 and 15.
A CLOSE statement which is entered without parameters, closes all the open files.

Characteristics — On execution of a CLOSE statement, the association between the file that was closed and its buffer is cancelled.
It is possible to re-use the buffer to open any other file.
A file that has been closed can be reopened through the use of an OPEN statement (in its own program range, or in that of another).
The OPEN statement can associate the same or another file number that is available at that moment.
The use of the CLOSE statement releases all the LOCKs connected to the closed files.

Example

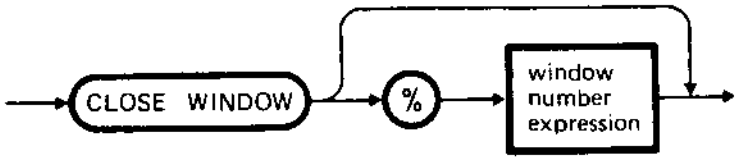
```
     .
     .
170 CLOSE # 2
     .
250 A=6
     .
290 CLOSE 3,5,A
     .
     .
     .
1200 CLOSE
```

Statement 170 closes the file associated to buffer number 2.
Statement 290 closes the files associated to buffer numbers 3,5 and 6 (if A is worth 6).
Statement 1200 closes all the files.

CLOSE WINDOW

Closes the specified window or all the open windows.

PROGRAM/IMMEDIATE Statement



where:

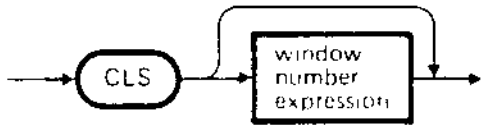
window number expression
: is a whole number expression, whose value identifies the window to be closed.
If window number expression is omitted, the system returns to the "initial state" (only one window: the full screen).

Characteristics
: The CLOSE WINDOW statement closes the window identified by the parameter window number expression. The system assigns the area of the window, which is closed, to the rectangle, which was split when it was opened.

The CLOSE WINDOW statement, if it is made to operate on the main window, does not have any effect. Window number 1 (or the main window) can never be closed.

Deletes the alphanumeric contents of the current window (or of a specified window) and positions the text cursor in the upper left-hand corner of the window itself.

PROGRAM/IMMEDIATE Statement



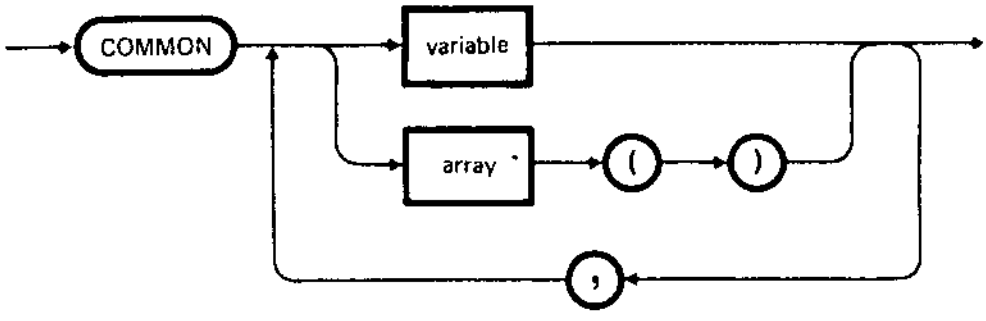
where:

window number expression — is an integer expression whose value identifies the window on which to operate. Use of this parameter is optional.
If it is not specified, the operation is executed on the current window.

COMMON

Allows the transfer of the "common" variables between two programs.

PROGRAM/IMMEDIATE Statement



Characteristics

COMMON statements are used with the CHAIN statement. A program can have one or more COMMON statements. It is not necessary for the chained program to specify, by means of COMMON statements, "common" variables that are specified by the chaining program. The chained program will refer to these variables by using the same names indicated in the chaining program.
Type-definition statements (DEFINT, DEFSNG, DEFDBL, DEFSTR) which may be associated to "common" variables, must precede COMMON statements and appear in both the chained as well as the chaining programs. (i.e. for the variable-types to be uniform.)
It is advisable not to write the same variable name in different COMMON statements within the same program (and not more than once in the same COMMON statement).
A COMMON statement can also specify array names. These names must be followed by a pair of

parentheses. Each array that is indicated in a COMMON statement, must be dimensioned with a DIM statement in the chaining program.
The DIM statement must precede the associated COMMON statement.

Example 1

```
10 REM PG1
20 COMMON A1,B1,C1,D1$
 .
 .
 .
80 CHAIN "PG2"
90 END
```

```
10 REM PG2
20 PRINT A1,B1,C1,D1$
 .
 .
 .
 .
. 120 END
```

In this example the values of the variables A1,B1,C1 and D1$ of the program PG1, are transferred to the chained program PG2 which displays their values (see statement 20).

Example 2

```
10 REM PG1
20 DEFDBL C
30 COMMON A1,B1,C1,D1$
 .
 .
 .
90 CHAIN "PG2"
100 END
```

```
10 REM PG2
20 DEFDBL C
 .
 .
 .
 .
 .
 .
130 END
```

The type definition statement DEFDBL, associated to a "common" variable, precedes the COMMON statement and must appear in the chained and chaining

programs.

Example 3

```
10 REM PG1
20 DIM A1(15,20)
30 COMMON A1(),B1,C1
 .
 .
 .
100 CHAIN "PG2"
110 END
```

```
10 REM PG2
 .
50 PRINT A1(1,1)
 .
90 END
```

The COMMON statement specifies an array name.
This name must be followed by a pair of parentheses.
The array indicated in the COMMON statement must be dimensioned with the DIM statement in the chaining program.
The DIM statement precedes the associated COMMON statement.

Resumes the execution of a program interrupted by a /CTRL/ /C/, or a BREAKON command, or a STOP, END or STON statement.

IMMEDIATE Command

Characteristics

Program execution is resumed from the point where the interruption occurred.
If the interruption took place after a prompt from an INPUT statement, program execution continues by reprinting the prompt (?).
Apart from the CONT command, it is possible to resume execution with the use of the GOTO statement in immediate mode. This passes execution control to a specified line number.
The command CONT will not allow the recovery of program execution if the latter has been modified or an error has occurred during the interruption.
The CONT command is normally used with the STOP, or STON/STOFF statements or with the BREAKON/BREAKOFF commands for program debugging.

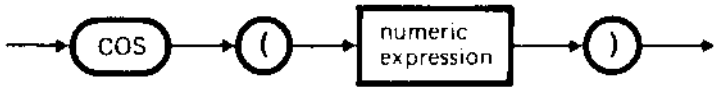
Example

```
10 INPUT A,B,C
20 K= 2*5.3:L=B-3/.26
30 STOP
40 M=C*K+100:PRINT M
RUN
? 1,2,3
Break in 30
OK
PRINT L
 30.7692
```

```
OK
CONT
 115.9
OK
```

COS (COSINE)

This function calculates the cosine of the argument.



Characteristics

The function argument is the size of an angle in radians. The cosine is calculated in double precision.

Note

The numeric expression can be of any type. The result provided is in double precision. It is advisable to use a double precision numeric expression as input.

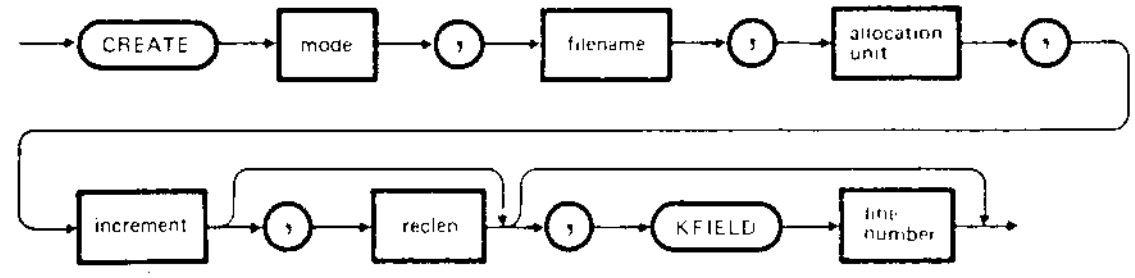
Example

```
PRINT COS(.4)
 .921060991671771
OK
```

CREATE

Creates a new file.

PROGRAM/IMMEDIATE Statement



where:

mode — is a constant, expression or string variable whose value may be:

- "A" (Append)
- "I" (Input)
- "O" (Output)
- "R" (Random)
- "K" (Keyed)

For further details see the OPEN statement.

filename — is a constant, expression or string variable which specifies a new file.
CREATE must be used when creating a keyed file. It is also advisable to use CREATE for the other types of files, when wishing to define their extent.

allocation unit	specifies the initial extent of the file. This must be expressed in number of sectors having 256 bytes each.
increment	specifies subsequent extentions in multiples of 256 bytes each.
reclen	specifies the record length (for files "R" and "K" files). If reclen is not specified, the default value is 256 bytes.
KFIELD	keyword. The use of this keyword is mandatory only for keyed files.
line number	specifies the FIELD statement in which the fields and keys (primary and secondary) are described.

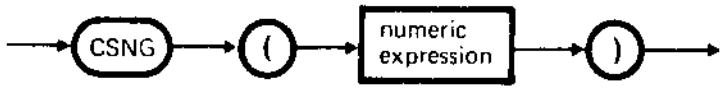
Example 1

```
CREATE "R", "file", 10,2
```

Example 2

```
100 CREATE "K", "F$", 100,10,50, KFIELD 120
110 OPEN "K", #1, F$
120 FIELD #1, 15 AS G$ KEY(1), 20 AS B$, 15
    AS C$ KEY(2)
```

This function converts a numeric-type argument to a number in single precision.



Note

To convert numbers to integer or double precision formats, see the CINT and CDBL functions.

Example

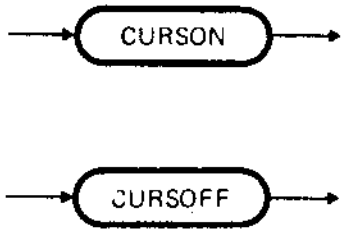
```
10 A# = 975.3421#
20 PRINT A# ; CSNG(A#)
RUN
 975.3421     975.342
OK
```

CURSOFF/CURSON (CURSOR OFF/CURSOR ON)

CURSOFF removes the text cursor from the screen.

CURSON displays the text cursor which has been removed by CURSOFF.

IMMEDIATE/PROGRAM Statements



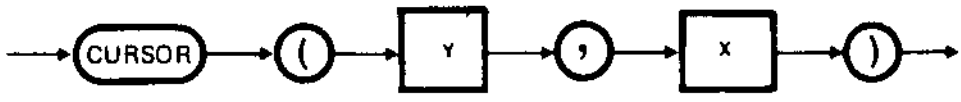
Note

The effect of the CURSOFF statement is annulled when exiting the BASIC environment.

CURSOR

Establishes the position of the text cursor on the screen.

PROGRAM/IMMEDIATE Statement



where:

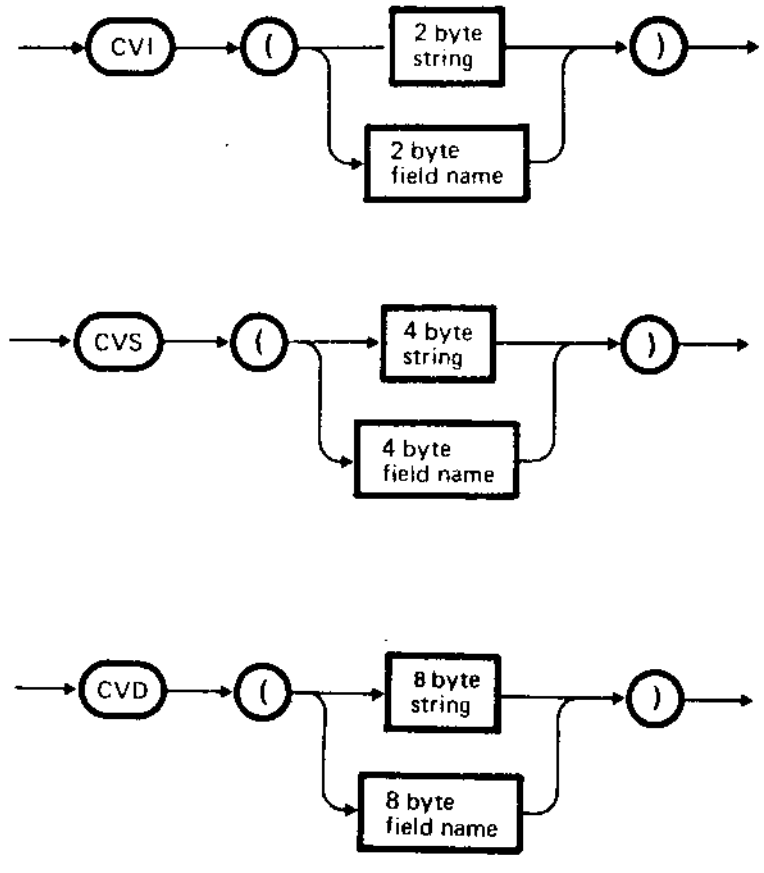
y,x — specify the position of the cursor inside the window; y and x are numeric expressions rounded up to the nearest integer and respectively provide the column and row positions.

Characteristics — The parameters y and x have a minimum value of 0. For y = 0 the column position assumed is the same as that of current position of the cursor; for x = 0 the row position assumed is the same as that of current position of the cursor.

CVI/CVS/CVD

These functions convert string values into numeric values.

- CVI converts a 2-character string into an integer
- CVS converts a 4-character string into a single precision number
- CVD converts an 8-character string into a double precision number



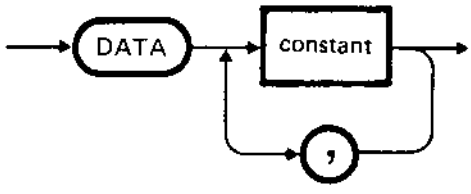
Examples

```
10    X# = CDV(N$)
20    Y! = CVS(R1$)
```

DATA

Creates an internal data file.

PROGRAM Statement.



Characteristics

The DATA statements create an "internal" file, that is, a sequence of numeric and string data, which must be transferred to the program variables by using one or more READ statements.
DATA statements can be placed at any point in the program.
Constant may contain numeric and/or string constants.
Numeric constants may have any type of format, i.e. integer, floating or fixed point (numeric expressions are not allowed). String constants must be quoted if they contain commas (e.g. DENVER,) colons or blanks at the beginning or at the end; in all other cases, use of inverted commas is unnecessary. Note that the data type in input (in the data sequence) must be of the same type as the variable type to which it is assigned, i.e. numeric variables require numeric constants as data, whereas, string variables require string data.

Examples

See the examples relative to the READ statement.

DATE$

This function allows the date to be entered and/or read.

PROGRAM/IMMEDIATE Statement

Characteristics

The date is set in terms of month, day, year (mm:dd:yyyy)
The user can make use of any ASCII character that can be printed (excluding numbers) as a delimiter.

In output slash always appears as the delimiter and the year is always expressed by 4 characters (in input the year may also be expressed by 2 characters).

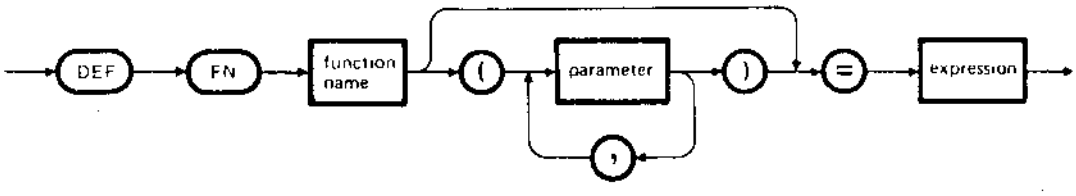
Example

```
100 IF DATE$="04:30:82"
    THEN 3000
     .
     .
     .
500 DATE$="05/06/1981"
     .
     .
     .
```

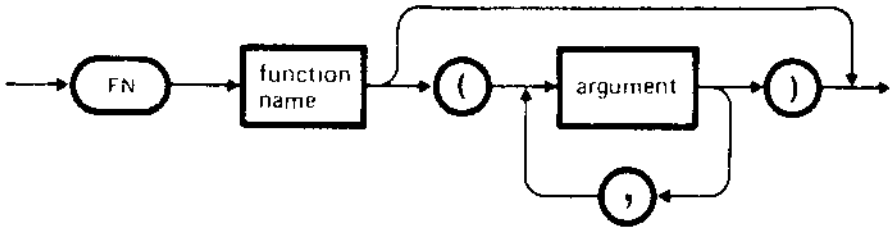
DEF FN (FUNCTION DEFINITION)

Defines a numeric or string type user-function. The DEF FN statement must be executed before the function that it defines is called.

PROGRAM Statement



Calling the Function



where:

function name — is a valid variable name beginning with FN (it is possible to specify numeric or string type names). No blanks may be inserted between FN and the name of the function. The first character of the name must be a letter.
If the function name specifies a name-type, the value of the function will be of the same type.

parameter — is one or more variables that must be replaced by the value(s) of the corresponding arguments when the function is called. The association between arguments/parameters is of a positional type (i.e. the first parameter corresponds to the first argument and so on).

argument — is the effective value attributed to the corresponding parameter. Each argument can be a constant, a variable, an expression or an array element.

expression — defines the operation to be executed by the function.
The expression can include variables defined outside the definition of the function (global variables). The parameter names in the expression define the function and do not affect in any way program variables with the same name. To render the program easy to read, the user should avoid giving the same name to parameters and variables. A variable name that is used to define a function, may or may not appear in the list of parameters. If it does appear, it is replaced as described in 'parameter', otherwise the current value of the variable is assumed.

Characteristics — If a user-defined function is invoked by another user-defined function, the function being invoked must be defined within the program itself in a previous position.

```
Example: 10 DEF FNA(X)=(SIN(X/5)*3.1)/180
         20 DEF FNB(X)=(FNA(X)+SIN(X))*.5
```

If a program is chained to another program through the MERGE option, the function definitions must be written in the chaining program before the CHAIN statement if they are called by the chained program. Otherwise, on completion of the operation MERGE, the user-defined function will be undefined.

```
Example: 10 DEF FNA(X)=(SIN(X/5)*3.1)/100
         20 DEF FNB(X)=(FNA(X)+SIN(X))*.5
         10 DEF FNC(X)=(X+X*(X+1))
```

```
.
.
.
100 CHAIN MERGE "PROG1"
```

Example

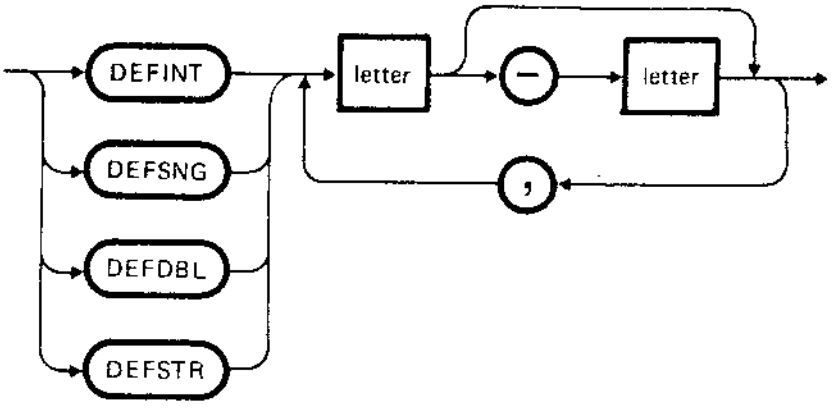
```
400 INPUT X,Y
410 DEF FNAB(X,Y)=X^3/Y^2
420 T=FNAB(X,Y)
```

Program line 410 defines the FNAB function.
Line 420 invokes the function.

DEFINT/SNG/DBL/STR

Declare the type of all the variables whose name begins with a specified letter.
These must be placed before the variables whose type they define and are normally inserted at the start of the program.

PROGRAM/IMMEDIATE Statements



Characteristics

DEFINT allows variables to be declared as integers. DEFSNG allows them to be declared in single precision. DEFDBL allows them to be declared in double precision. DEFSTR allows them to be declared as strings. If the user over-rides the type of a variable by adding a type-definition tag at the end of the name, this operation has precedence over a DEF type statement. If no type definition statement is carried out, BASIC will consider all variables without type-definition tags in single precision. If a variable is defined twice, the last definition is valid.

Example 1

```
10 DEFINT A-Z
```

All the program variables are integers.

Example 2

```
10 DEFDBL D
```

All the program variables beginning with the letter D are in double precision.

Example 3

```
10 DEFSTR S,U-W
```

All the program variables beginning with the letters S, U, V and W are string variables.

Example 4

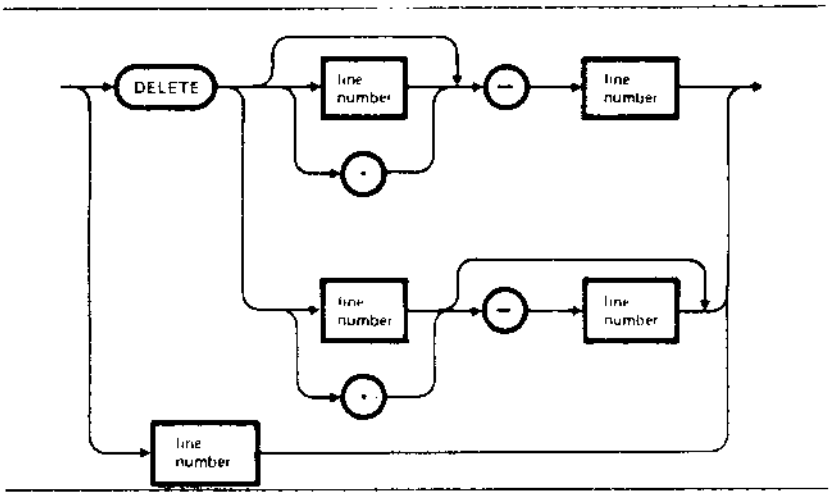
```
10 DEFINT A-Z
20 DEFDBL D
```

All program variables are integers except for variables beginning with the letter D which are in double precision.

DELETE

Deletes the program lines.

IMMEDIATE Command



Example 1

DELETE .

The current line is deleted.

Example 2

500

or

DELETE 500

Line 500 is deleted.

Example 3

DELETE 100-200

All the lines between 100 and 200 both numbers inclusive are deleted.

Example 4

```
DELETE -400
```

All the lines from the beginning of the program to line 400 inclusive are deleted.

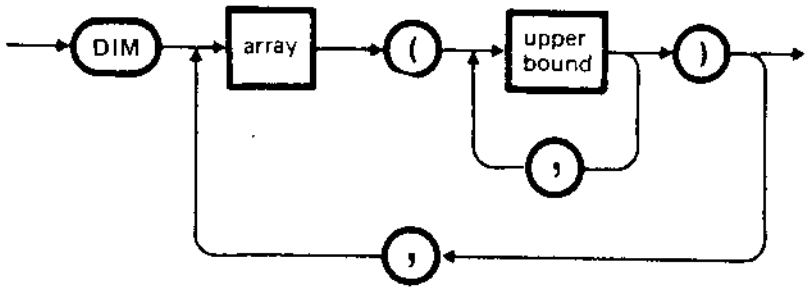
Example 5

```
DELETE .-500
```

All the lines starting from the current line to line 500 inclusive are deleted.

Specifies the name of one or more arrays, the dimension numbers of each array and the maximum index value for each dimension.
Moreover, it initialises all the elements of the specified arrays to zero and allocates the space in memory for the array.

PROGRAM Statement



where:

array — is a array name (any variable name is allowed).

upper bound — is any positive numeric constant or numeric variable with a positive value. If this value is not whole it is rounded up to the nearest integer. It establishes the maximum index value for each dimension. The number of dimensions is determined by the number of upper bounds provided .

Characteristics

In BASIC it is possible to define arrays with a maximum number of dimensions equal to 10.
The minimum index value for each dimension is 0, unless otherwise established through the use of the OPTION BASE statements.
If the user does not insert a DIM statement in the program, the array is created when BASIC encounters one of its elements for the first time. On the basis of the number of indices, the number of dimensions is determined and the maximum index value is 10. For example, if the following appears in a statement:

```
AR1 (3,5,10)
```

the array AR1 is created with three dimensions and with a maximum index value for each dimension equal to 10.

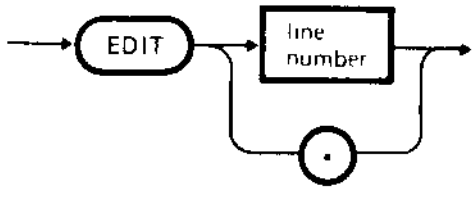
Example

```
10 DIM A(5), B$(20,30)
```

Defines array A as a one-dimensional numeric array with an index between 0 and 5, (both numbers inclusive), and array B$as a two-dimensional string array with its row index between 0 and 20 (both numbers inclusive) and its column index between 0 and 30 (both numbers inclusive).

Causes the system to enter the Editor State at the specified line.

IMMMEDIATE Command



Characteristics

In the Editor State it is possible to modify parts of a line. The following keys can be used to modify the line of a program:

--> moves the cursor to the right of its current position.
If the cursor is already at the end of the line, the command is ineffective.

<-- moves the cursor to the left of its current position.
If the cursor is already at the beginning of the line, the command is ineffective.

-->| moves the cursor to the end of the line.

|<-- moves the cursor to the beginning of the line.

K--| or BS the cursor shifts one position to the left and the character on which it was positioned, is deleted.

DC	the character on which the cursor is positioned, is deleted; all the characters on the right shift one space to the left. The cursor position remains the same.
IC	enters the Insert State. Any character that is keyed in this State is inserted in correspondence with the cursor position and will cause all the characters to shift one space to the right of the cursor. Even the cursor position will be shifted one space to the right. During the Insert State all the keys described above may be used. It is possible to exit from the Insert State by entering IC.
RESET	resets all characters to zero starting from the cursor position to the end of the line.
any graphic character	in Normal State, is displayed in correspondence with the cursor position, thereby replacing the previous character (tape over)
end of input key	saves all the changes made on the line and allows exit from the Editor State.
/CTRL/ /C/	the system returns to the Command State. Any change made whilst in the Editor State is lost.
↓	Changes made on the line are saved. The Editor State will operate on the next line.
↑	Changes made on the line are saved. The Editor State will operate on the previous line.
Note	If a Syntax Error occurs during program execution, the Editor State automatically enters on the line that caused the error. When changes to the line have been made and the end of input key has been entered, the line is reinserted, however, this causes all the variable values to be lost.

END

Terminates program execution and closes all the data files

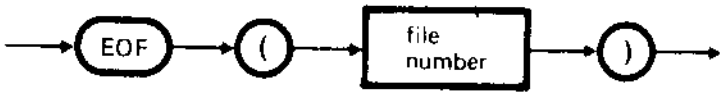
PROGRAM Statement

Characteristics

It is unnecessary to terminate a program with an END statement, however, it is useful as it closes all the open data files and enhances program readability. The END statement can be inserted in any part of a program to terminate its execution. It is also used to terminate a program at the end of a branch. Execution of an END statement always causes a return to the Command State

EOF (END OF FILE)

This function provides the true value if the end of a sequential file has been reached.



where:

file number — is a numeric expression rounded up to the nearest integer, which represents the file numbe

Note — It is advisable to use the function EOF during the read operation of a sequential file, in order to verify whether the end of the file has been reached.

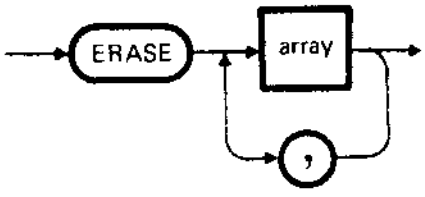
Example

```
10 DIM A(50)
20 OPEN "I",1,"DATA"
30 FOR K%=0 TO 50
40 IF EOF(1) THEN 100
50 INPUT #1,A(K%)
60 NEXT K%
```

ERASE

Frees the space reserved for one or more arrays and makes the names associated to them available for other program variables.

PROGRAM Statement



Characteristics

After the execution of an ERASE statement, the arrays can be redimensioned.

Example

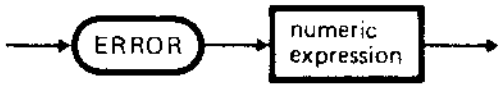
```
10 DIM A(15,15),B(10,20)
    .
    .
    .
100 ERASE A,B
110 DIM A(100),B(2,2,2)
```

After having executed statement 100, the space previously allocated to arrays A and B is released. The user can then define other variables; (arrays in particular) with the same names. Statement 110 defines two other arrays A and B with a different dimension number and with a different maximum index value.

ERROR

Simulates the occurrence of an error in the BASIC language or generates a user-defined error.

PROGRAM/IMMEDIATE Statement



where:

numeric expression — the value of the numeric expression represents an error code. It must be greater than 0 and less than 255. If it is not an integer, it is rounded up to the nearest integer.

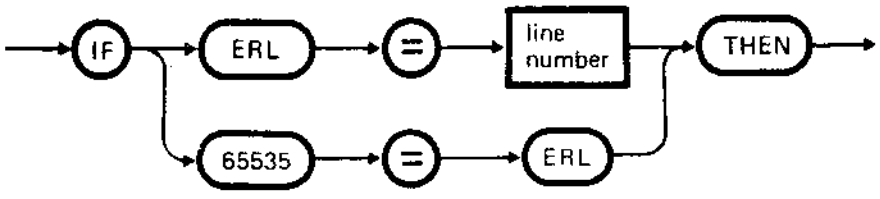
Characteristics — If the value of numeric expression corresponds to a BASIC error code (see Appendix A), the ERROR statement simulates the occurrence of this error and the corresponding error message will be displayed. If the value of numeric expression is greater than all the BASIC error codes, the ERROR statement generates a user-defined error. This error code can be conveniently used later in an error handling routine (see ON ERROR GOTO). In order to define his own error codes, the user must make use of a value that is greater than all the BASIC error codes. If an ERROR statement specifies a code for which no error message was defined, BASIC will reply with the following message:

Unprintable error

Example

```
10 S=10
20 T=5
30 ERROR S+T
40 END
OK
String too long in line 30
OK
ERROR 15
String too long
OK
```

ERL returns the line number in which an error has occurred. ERR returns the error code.



Characteristics

The functions ERL and ERR are normally used in the statements IF...THEN ...ELSE or IF...GOTO...ELSE so as to allow the user to handle the error. If the statement causing the error was an immediate statement, ERL will contain the value 65535. To check whether an error has effectively taken place in an immediate statement, use:

```
IF 65535 = ERL THEN...
```

In the other cases use:

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

Note

If the line number is not found on the right of the relational operator, it cannot be renumbered with the RENUM statement.

Example 1

```
10 REM RECTANGLE2
20 ON ERROR GOTO 70
30 INPUT "Base and Height";B,H
40 IF (B<0) OR (H<0) THEN ERROR 200
50 PRINT "Area=";"B*H:" B=";B;" H=";H
60 GOTO 30
70 IF (ERR=200) AND (ERL =40)
   THEN PRINT "B or H<0";RESUME 30
90 END
OK
RUN
Base and Height? -2,5
B or H<0
Base and Height? 2,5
Area= 10 B-2 H-5
Base and Height? ^ C
Break in 30
OK
```

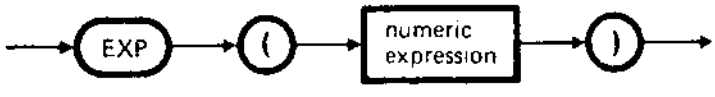
If a negative value is given for B or for H, the error handling routine is activated and the system displays:

```
B or H<0
```

Execution is resumed at statement 30 (see the RESUME statement).

EXP (EXPONENT)

This function raises the power of the constant 'e'. The exponent of the power is the value of the argument.



Characteristics

the value of the argument must be less than (709.782712893384). Computation of EXP is carried out in double precision.

Note

The numeric expression can be of any type. The result provided is in double precision. It is advisable to use a double precision numeric expression as input.

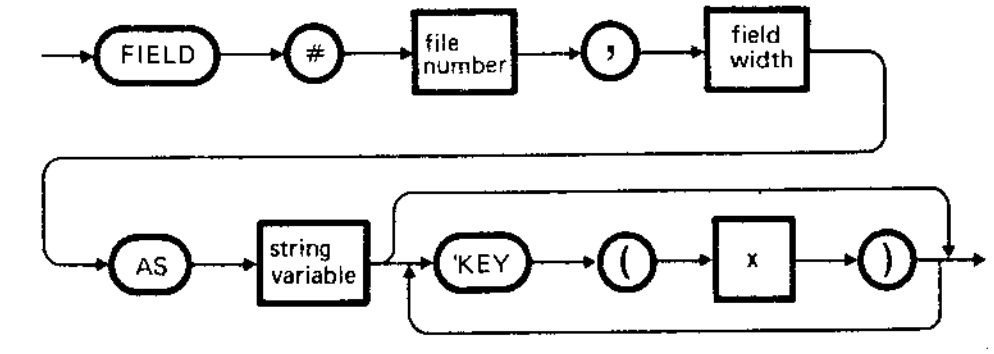
Example

```
10 X=5
20 PRINT EXP(X-1)
RUN
 54.5981500331438
OK
```

FIELD

Assigns space in the buffer of a random file to the specified variables.
Assigns space in the buffer of a keyed file to the specified variables, moreover, specifies the fields to be used as keys.

PROGRAM/IMMEDIATE Statement



where:

file number	is a numeric expression whose rounded value specifies the file number.
field width	is the number of bytes to be allocated in the field. One byte corresponds to one character.
string variable	is a string variable to be used as a field to specify the primary and secondary keys.
KEY	is an optional keyword which is only used for keyed files and it specifies which string variable is identified as a key in the operations GET, PUT, REMOVE. The parameter x is an integer that can assume the values from 1 to 5 with the following meaning:

KEY(1) identifies a primary key
KEY(2) identifies a secondary key
KEY(3) identifies a secondary key
KEY(4) identifies a secondary key
KEY(5) identifies a secondary key

Characteristics

The total number of bytes allocated through the use of the statement FIELD must not exceed the record length which is specified when the file is opened (the default record length is 256 bytes).
Any number of FIELD statements can be executed on the same file and they enter into operation concurrently.
Note that all the KEY parameters must be specified in a single FIELD statement.

Note

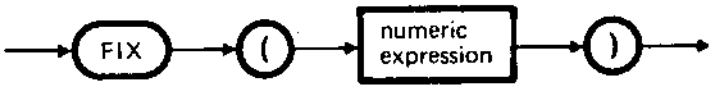
Do not use a variable name that is used in a FIELD statement in a LET or INPUT statement.

Example

```
FIELD 1,20 AS N$ KEY(2), 10 AS ID$, 40 AS ADD$
KEY(1)
```

The first 20 positions in the buffer of the keyed file are assigned to the string variable N$, which can be used as a secondary key in a GET operation.
The 10 positions that follow are assigned to ID$ and the following 40 to ADD$, which can be used as a primary key in the operations GET/PUT/REMOVE.

This function provides the integer part of the argument.



Characteristics

FIX(X) is equivalent to SGN(X)* INT(ABS(X)).
By comparison to INT, FIX does not provide the immediately lower value, in the case of negative arguments.

Examples

```
PRINT FIX(58.75)
 58
OK
PRINT FIX(-59.75)
-58
OK
```

Specifies how to represent the display of numeric data.

PROGRAM/IMMEDIATE Statement

where:

format type — this parameter indicates the type of editing used to display or print numeric data. It can assume one of the following three values:

I (English)

E (European)

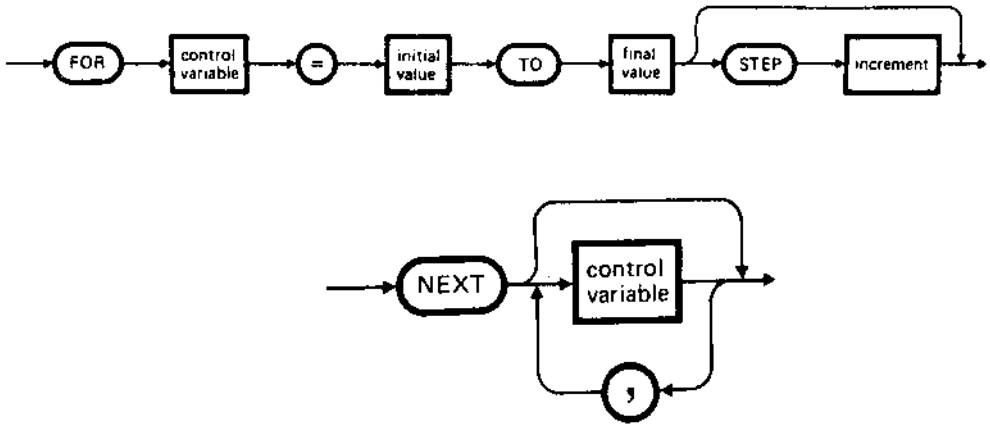
A (American)

filling character — is any printable ASCII character with which the field which contains the number will be filled.

FOR / NEXT

The statements FOR/NEXT allow a series of statements to be executed in a loop for a certain number of times.

PROGRAM/IMMEDIATE Statement



where:

control variable — is a simple numeric variable (which can be an integer or in single precision). The control variable names specified in the FOR and NEXT statements must be equal. A list of control variables can follow the word NEXT, however it is also possible to write a NEXT statement on its own. If no control variable follows the word NEXT, the NEXT statement will be automatically associated with the last FOR statement executed.

initial value	is a numeric expression specifying the initial value which is assigned to the control variable when a FOR statement is executed.
final value	is a numeric expression specifying the control variable value limit. This value is compared with the control variable every time the loop is repeated.
increment	is a numeric expression specifying the increment, that is, the value to be added (with its algebraic sign) to the control variable every time the NEXT statement is carried out. If the STEP option is not specified, an +1 increment is assumed.

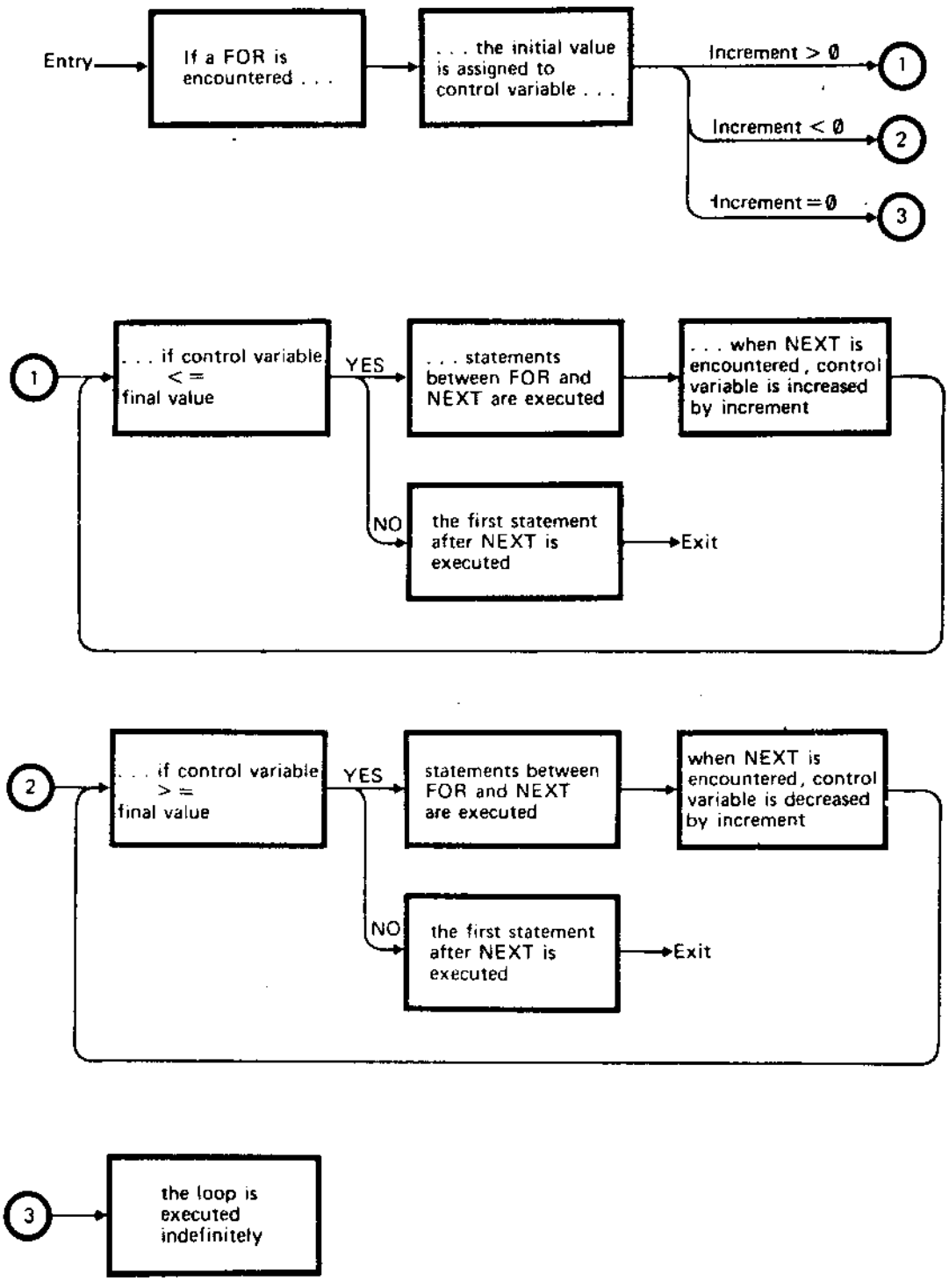


Fig. 1 - Function of the FOR/NEXT Statements

(*) Unless the initial and final values coincide. In this case the first statement after NEXT is carried out.

Characteristics

Increment of a Positive Value.
If the value of the increment is positive, the FOR/NEXT loop is executed until the value of the control variable is not greater than the final value.

Example:

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

In this case, the loop is repeated five times.
If the value of the increment is positive and if the initial value is higher than the final one, the loop is not executed.

Example:

```
10 J=0
20 FOR I=1 TO J
30 PRINT I
40 NEXT I
50 PRINT "Exit of the loop"
OK
RUN
Exit of the loop
OK
```

Increment of a Negative Value.
If the value of the increment is negative, the loop is carried out until the value of the control variable is not less than the final value.

Example:

```
10 FOR I%=1 TO -10 STEP -3
20 PRINT I%;
30 NEXT I%
40 PRINT "Exit";
50 PRINT " CONTROL VARIABLE=";I%
OK
RUN

1
-2
-5
-8
 Exit CONTROL VARIABLE=-11
```

In this case, the loop is executed four times.
When it is terminated, the control variable keeps the last value that has been assumed (-11) which is displayed with statement 40.
If the value of the increment is negative and if the initial value is less than the final one, the loop is not carried out.

Example:

```
10 FOR K%=1 TO 10 STEP -2
20 PRINT K%
30 NEXT K%
40 PRINT "Exit"
50 PRINT " CONTROL VARIABLE=";K%
OK
RUN
Exit CONTROL VARIABLE=1
OK
```

Increment of a Value equal to Zero.
If the value of the increment is equal to zero, the loop is repeated an infinite number of times (unless the initial and the final values coincide in this case, the loop is not executed).

Example:

```
100 FOR A%=1 TO 30 STEP 0
110 PRINT A%
120 NEXT A%
OK
RUN

1
```

```
1
.
.
.
```

The user must enter /CTRL/ /C/ to interrupt execution.

Nested loops.
Two or more FOR/NEXT loops can be nested on condition that the internal FOR/NEXT loop is completely included in the outside one. The following loops, for example, are correct:

```
50 FOR I = 1 TO 10
100 FOR J = 2 TO 20
200 NEXT J
300 NEXT I
```

whereas the following are not:

```
50 FOR I = 1 TO 10
100 FOR J = 2 TO 20
150 NEXT I
200 NEXT J
```

Two or more FOR/NEXT nested loops must not have the same control variable.
For every FOR statement, there must be a corresponding NEXT statement.
If all nested loops have the same final period, a single NEXT statement, together with a list of all the control variables separated by a comma may be used to group all loops.
In a series of nested FOR statements, if one or more NEXT statements are missing, the error 'FOR without NEXT' is always signalled on the first FOR even if it is correctly closed with the relative NEXT.

Example

```
10 REM PRIME NUMBERS
20 INPUT "Enter limits N,M";N,M
30 PRINT "Primes from";N;"TO";M
40 PRINT
50 PRINT
60 FOR I=N TO M
70 LET K=SQR(I)
80 FOR J=2 TO K
90 LET E=I/J-INT(I/J)
100 IF E=0 THEN 130
110 NEXT J
120 PRINT I;
```

```
130 NEXT I
140 PRINT
150 PRINT
160 PRINT "End of List"
170 END
OK
RUN
Enter limits N,M? 1,15
Primes from 1 to 15

1 2 3 5 7 11 13
End of List
OK
```

All prime numbers included together in one data item are displayed. A FOR/NEXT loop specifies the group of number under consideration. A second loop nested in the first contains an algorithm to determine which numbers of the specified group are prime numbers.

To explain the algorithm: the numbers assigned to a variable (in this case I) are different for an integer (in this case J), whose value is included between 2 and the square root of I. If the remainder of the division is 0, I is not a prime number. The number I+1 is then generated and execution is repeated. The square root of the final value is chosen because if there are whole factors of the number I, these will always be placed between 2 and the square root of I.

Statement 100 allows the exit from the internal loop even though J is not greater than K. It is possible to exit from a loop when a specified condition has been satisfied.
However it is not possible to enter in the "middle" of a loop.

If the increment is equal to zero and the initial and final values concide, then the loop is not executed and control is passed to the first executable statement after the NEXT.
A FOR/NEXT loop can be defined as pending if it is interrupted by a "break" before its termination. Any change made to the resident program (e.g. deletion and modification of lines) will prevent the loop from terminating.

The final value is always defined before the initial

one. If for example, the following is inserted:

```
10 I-5
20 FOR I=1 TO I+5
```

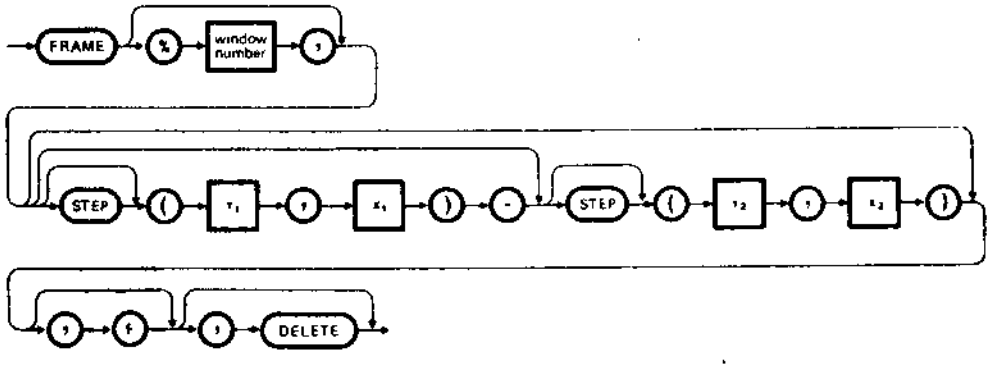
statement 20 assigns the value 10 to the final value.
For greater program readability, it is unadvisable to use the control variable to define the final value.

It is advisable to use an integer variable for the control variable and integer constants (or variables) for the initial and final values and for the increment. This optimizes execution time.

FRAME

Outlines the frame of a specified window or of a section of it.

PROGRAM/IMMEDIATE Statement



where:

window number — is aninteger numeric expression which specifies and selects the window on which the FRAME statement must operate.
If omitted, it will operate on the current window.

STEP — is an optional keyword. Allows the use of relative coordinates. Both the initial (y1,x1) and final coordinates (y2,x2) become relative to the current position of the text cursor through the use of STEP.

y1,x1 — are the coordinates of the initial point of the frame (these coincide with the upper left angle of the frame itself). If omitted, the frame starts at the current position of the cursor. Both y1 and x1 provide the column and row position of the given point. This is the same for y2 and x2.

y2,x2 — are the coordinates of the final point of the frame (these coincide with the lower right corner of the frame itself). If the co-ordinates (y1,x1) and the co-ordinates (y2,x2) are omitted, the whole window is re-framed.

F (filled) — is an optional parameter. Through the use of the parameter F, the selected area is filled with the colour used to draw the outline.

DELETE — is an optional keyword. Use of DELETE cancels the effects specified by the other FRAME options

Characteristics — By simply entering the keyword FRAME the current or the specified window is outlined.
If the value of the parameter x1 coincides with that of the parameter x2 the outline is reduced to a vertical line. If, however, y1 concides with y2, a horizontal line is displayed.

Example 1 — FRAME (16,13) - (25,21), F

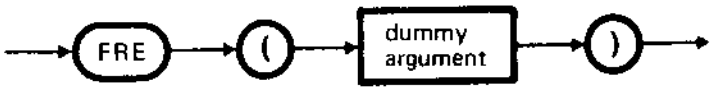
This statement outlines a portion of the current window. The upper left-hand and the lower right-hand corners both have the co-ordinates (16,13) and (25,21) respectively. The delimited area is filled with the colour used to draw the outline.

Example 2 — FRAME STEP(0,10) - STEP(5,21)

This statement executes the operation described in example 1. The only difference is that the co-ordinates are relative to the current position of the cursor and the delimited area is not filled.

FRE (FREE SPACE)

This function calculates the memory space that is not used by the BASIC program.



where:

dummy argument — is a numeric or a string expression. The value calculated by the function is the same no matter what argument is given.

Characteristics — FRE(""), before calculating the number of bytes available, forces a "garbage collection" (the elimination of empty spaces). This operation is executed automatically should the user memory be saturated.

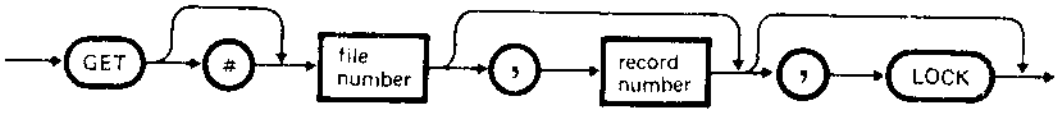
Examples

```
PRINT FRE(0)
 14542
OK
PRINT FRE(X$)
 14542
OK
```

Format 1

Reads a record from a random file.

PROGRAM/IMMEDIATE Statement.



where:

file number — is a numeric expression whose rounded value specifies the file number.

record number — is a numeric expression whose rounded value specifies the number of the record to be read. If it is omitted, the current record is read.

The minimum number of records is 1, the maximum is 32767.

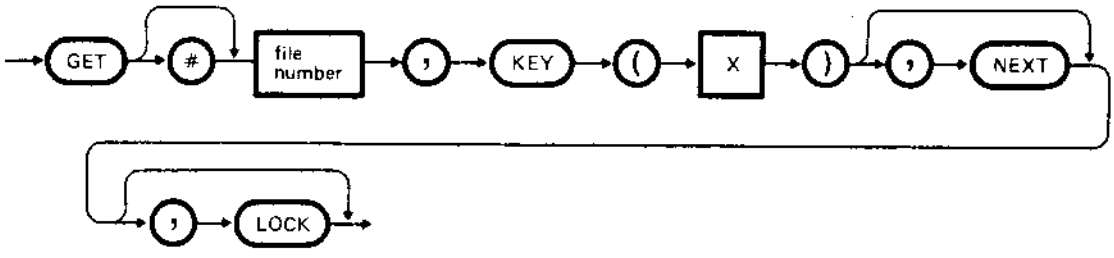
The current record is the record whose number is greater than the last record selected by one. The first time the user accesses a random file, (without specifying the record number), the current record number is defined as equal to 1.

LOCK — optional keyword which, if specified, prevents any other user to update the specified record. If the LOCK clause has already been set on the record, the GET statement displays an error message.

Format 2

Reads a record from a keyed file.

PROGRAM/IMMEDIATE Statement



where:

file number	is a numeric expression whose rounded value specifies the file number.
KEY	keyword specifying which string variable is identified as a key. Parameter X is a number that can assume integer values from 1 to 5 with the following meaning: KEY (1) identifies the primary key KEY (2) identifies a secondary key KEY (3) identifies a secondary key KEY (4) identifies a secondary key KEY (5) identifies a secondary key The value of KEY (X) must be provided by the variable specified as KEY (X) in the FIELD statement.
NEXT	optional keyword allowing the subsequent record to be read.
LOCK	optional keyword preventing any other user to update the specified record. If the LOCK clause has already been included in the record, the GET statement will have no effect on the LOCK statement.

Example 1

```
 10 FIELD #1,20 AS N$ KEY(1), 4 AS A$, 30 AS P$ KEY(2)
 20 LSET N$="SMITH"
 30 GET #1, KEY(1)
    .....
    .....
100 LSET P$="CHICAGO"
110 GET #1, KEY(2)
    .....
    .....
200 LSET N$="SMITH"
210 GET #1, KEY(1) , NEXT
```

The identified record of the primary key "SMITH" in the statements 10, 20, and 30 are transferred to the buffer of the keyed file.

The record that is identified by the secondary key "CHICAGO" in statements 100 and 110 is transferred to the buffer of the keyed file. The record with a key that is greater than "SMITH" in statements 200 and 210 is transferred to the buffer of the keyed file through the use of the primary index.

Example 2

```
   ...
   ...
70 OPEN "k",#1,F$,20
80 FIELD #1,5 AS A$ KEY(1),5 AS B$ KEY(2),10 AS C$
90 INPUT "0=read 3=exit",K$
100 IF K$="3"THEN 390
    ...
    ...
180 INPUT "0=key prim 2=key sec"; Z$
190 INPUT "0=next 1=key"; U$
200 IF U$="0" THEN 290
210 INPUT "key input 11=5"; AA$
220 IF Z$="0" THEN 260
230 LSET B$=AA$
240 GET#1,KEY(2)
250 GOTO 330
260 LSET A$=AA$
270 GET#1,KEY(1)
280 GOTO 330
290 IF Z$="0" THEN 320
300 GET#1,KEY(2),NEXT
310 GOTO 330
320 GET#1,KEY(1),NEXT
330 PRINT A$,B$,C$
340 GOTO 90
    ...
    ...
390 CLOSE
400 END
```

Statement 70 opens a keyed data file identified by the contents of a string variable F$. The length of each record is 20 bytes and the file number is 1. Statement 80 divides the buffer in fields which are used to specify primary and secondary keys. With statement 90 the program accepts input from keyboard and consequently can execute read operations or terminate the program.

GOSUB/RETURN

GOSUB calls a subroutine by passing control to the first line number of the subroutine. RETURN transfers control to the first statement following the last GOSUB executed.

PROGRAM Statement

---

where:

line number — is the first line number of a subroutine.

Characteristics — A subroutine can begin with any statement (excluding NEXT). It is recommended to always begin a subroutine with REM (or with a statement that terminates with a comments field).
A subroutine can end with a RETURN statement, which must be executed last, since it is the only statement which allows control to be passed back to the main program.

A subroutine can have more than one RETURN statement when it is structured so as to have several branches, each of which request that control be

passed back to the main program.
A subroutine can be called in any point of the program, any amount of times, if a program calls one of its own subroutines more than once when the subroutine has been executed control is passed on to the statement following the GOSUB executed last.

A subroutine can be inserted at any point in the program, however, it is recommended to write subroutines one after another, at the end of the program and close the program (before the start of the first subroutine) with either an END or GOTO or STOP statement.

A subroutine can call another subroutine. The number of "nested" subroutines that may be active at the same time is limited by available memory.

A subroutine can access each program variable. Infact, all the variables that are defined in the "main" program are also known to the subroutine. Thus, these can operate without restriction on each variable (and also change their value).

Example 1

```
10 DEFINT A-Z 'defines all the variable integers
20 INPUT "Enter 3 integers"; A,B,C
30 LET X=A
40 LET Y=B
50 GOSUB 110
60 LET X=M
70 LET Y=C
80 GOSUB 110
90 PRINT "MCD of";A;B;C"=";M
100 GOTO 190
110 LET Q=INT(X/Y) 'subroutine which calculates
    the MCD of X and Y.
120 LET R=X-Q*Y
130 IF R=Q THEN 170
140 LET X=Y
150 LET Y=R
160 GOTO 110
170 LET M=Y
180 RETURN
190 END
OK
Enter 3 integers? 1377,2916,405
MCD of 1377 2916 405 = 81
OK
RUN
Enter 3 integers? 4,3333,67
MCD of 4 3333 67 = 1
OK
```

This program shows the use of a subroutine.

The subroutine makes use of Euclydes algorithm to find the maximum common denominator (MCD) of three integers (A,B and C). These are entered the first two, A and B are assigned to the variables X and Y respectively (see statements 30 and 40) and the subroutine determines their MCD (see statements 110 to 180). The MCD that is found, is assigned to variable X in statement 60 and the third number (C) is assigned to variable Y in statement 70.

The subroutine is again invoked (see statement 80) to find the MCD of these two numbers.

The result is the MCD of the three numbers. These are displayed together with their MCD using statement 90.

Statement 10 defines all the integer variables, as the program only operates on integers.

Example 2

```
10 INPUT "Enter N>0";N%
20 IF N%<=0 THEN 10
30 GOSUB 50
40 END
50 REM SUB1 (Sum of Integers)
60 S%=(N%*(N%+1))/2
70 PRINT "Sum of Integers from 1
         to "N%;"=";S%
80 INPUT "Sum of Squared Numbers
         (S/N)";X$
90 IF X$="S" THEN GOSUB 110
100 RETURN
110 REM SUB2 (Sum of Squared Numbers)
120 S2%=(N%*(N%+1)*(2*N%+1))/6
130 PRINT "Sum of Squared Numbers from
         1 to ";N%;"=";S2%
140 RETRUN
OK
RUN
Enter N>0? 5
Sum of Integers from 1 to 5= 15
Sum of Squared Numbers (S/N)? S
Sum of Squared Numbers from 1 to 5=55
OK
```

This program calculates the sum of integers from 1 to N (where N is keyed in), and if requested, the sum of the squared numbers.

The program has two subroutines SUB1 and SUB2 that are inserted at the end (statements from 50 to 100 and from 110 to 140)

Statements 10, 20 and 30 are executed first. Statement 30 (GOSUB) calls the SUB1 subroutine and the statements of the latter are executed sequentially up to statement 90.

This carries out a test:

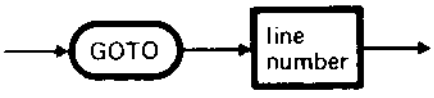
- if the value of X$ (which is entered) is different from "S", control is passed to statement 100 (RETURN) and therefore to statement 40 (END).

- if the value of X$ is equal to "S", the SUB2 subroutine (which is "nested") is called. When statement 140 is reached, (RETURN in SUB2), control is passed to statement 100 (RETURN in SUB1) and therefore to statement 40 (END).

GOTO

Passes control to the specified program line.

PROGRAM/IMMEDIATE Statement



Characteristics

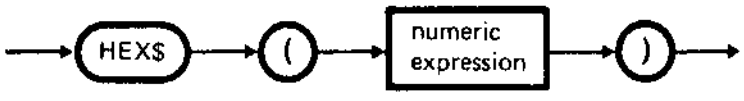
If the statement specified by GOTO is not executable (e.g. it is a REM statement), control is passed to the first executable statement following the specified line number. Use of the GOTO statement in Command State ensures that program execution begins at the specified line number. This allows values to be assigned to program variables in the Command State. This technique can be used in the program debugging phase.

Example

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

Statement 50 passes unconditional control to statement 10.

This function converts the decimal argument into its corresponding hexadecimal value according to the ASCII table.



where:

numeric expression — is a numeric expression that is rounded to the nearest integer.

Note — See the function OCT$ for octal conversion.

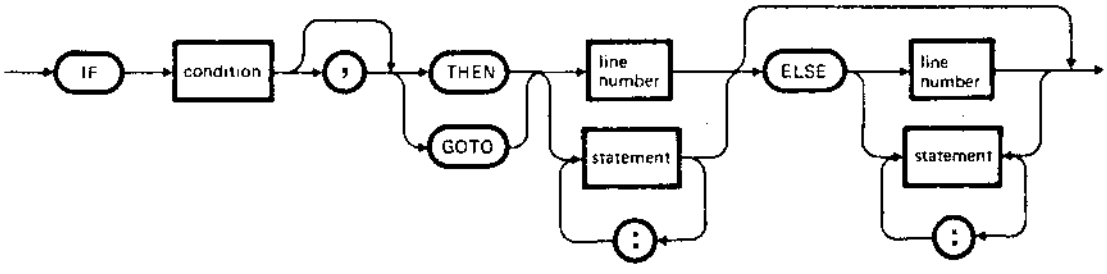
Example

```
10 INPUT X
20 A$ = HEX$(X)
30 PRINT X "DECIMAL IS " A$ " HEXADECIMAL"
RUN
? 32
 32 DECIMAL IS 20 HEXADECIMAL
OK
```

IF...GOTO...ELSE
IF...THEN...ELSE

Both these statements pass control, conditionally, to a specified statement.

PROGRAM/IMMEDIATE Statement



where:

condition — can be one of the following expressions:

- numeric
- relational
- logical

Characteristics — BASIC determines whether the condition is either true or false, by checking if the result (numeric) is different from zero (true condition) or equal to zero (false condition). For this reason, it is possible to check whether the value of a variable is equal to, or different from zero, by specifying the variable name as a "condition". If the condition is true, control is passed to the statement whose line number is specified after GOTO (or THEN), or to the

first statement after THEN.
If the condition is false and the ELSE clause is omitted, control is passed to the first executable statement following the statement IF... GOTO or IF...THEN.
If the condition is false and the clause ELSE is included, control is passed to the statement whose line number is specified after ELSE or to the first statement that follows ELSE.
After executing out the statement (or the statements) that follows ELSE, control is passed to the first executable statement.

The IF...GOTO...ELSE or IF...THEN...ELSE statements may be nested.
Nesting is limited only by the length of the BASIC line (255 characters).
If an IF...GOTO...ELSE or an IF...THEN...ELSE statement is used to check the result of a floating point computation, as the internal representation of the value may not be exact, checking is carried out in the range of the accuracy defined.
For example to test whether variable A is equal to 1.0 use.

```
IF ABS(A-1.0)<1.0E-6 GOTO
```

or

```
IF ABS(A-1.0)<1.0E6 THEN...
```

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

Example 1

```
10 REM IF GOTO test program
20 INPUT X%
30 IF X%>=10 GOTO 60
40 PRINT "IF GOTO failed the test"
50 GOTO 99
60 PRINT "IF GOTO passed the test"
99 GOTO 20
OK
RUN
? 10
IF GOTO passed the test
? 2
IF GOTO failed the test
? ^ C
Break  in 20
OK
```

If 10 is entered, the condition of statement 30

(X%=10) is true and control is passed to statement 60. If 2 is entered, the condition is false and control is passed to statement 40.

Example 2

```
10 INPUT X
20 IF X= INT(X)
   THEN PRINT X; "is an integer"
   ELSE PRINT X; "is not an integer"
30 IF X= 9999 THEN END ELSE 10
OK
RUN
? 1
 1 is an integer
? 1. 5
 1.5 is not an integer
? ^ C
Break in 10
OK
```

If 1 is entered, the condition (X=INT(X)) indicated in statement 20 is true and control is passed to the PRINT statement that follows THEN. If 1.5 is entered, the condition is false and control is transferred to the PRINT statement that follows ELSE.
Statement 20 is a single logical line divided into three physical lines.

Example 3

```
50 IF I THEN A=1000
```

1000 is assigned to variable A if I is not equal to 0.

Example 4

```
70 IF (I<30) AND (I>5) THEN
   A=B+C: GOTO 350
80 PRINT "OUT OF RANGE"
      .
      .
      .
```

A test determines whether I is greater than 5 and less than 30. If I is in this range the value of A is calculated and execution passes to line 350. If I is not in this range, execution continues from line 80.

INKEY$

This function calculates a one character string: the character is entered from keyboard. A null string is returned if there is no character waiting to be processed. No characters will be echoed and all characters are passed to the program except for /CTRL/ /C/ which interrupts program execution.

Example

```
1000'Timed Input Subroutine
1010 RESPONSE$=''''
1020 FOR I%=1 TO TIMELIMIT%
1030 A$=INKEY$:IF LEN(A$)=0 THEN 1060
1040 IF ASC(A$)=13 THEN TIMEOUT%=0:RETURN
1050 RESPONSE$=RESPONSE$+A$
1060 NEXT I%
1070 TIMEOUT%=1:RETURN
```

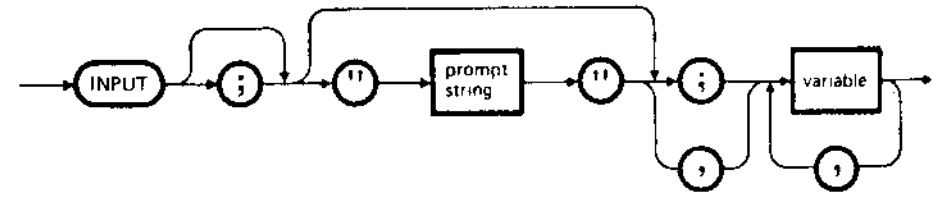
This routine returns two values:

- RESPONSE$ that contains the string entered
- TIMEOUT% which is equal to zero if the user enters a string of characters from keyboard before completing a specified number of FOR/NEXT loops (a number equal to TIMELIMIT%), otherwise equal to 1.

INPUT

Reads data from the keyboard and assigns them to one or more specified variables.

PROGRAM Statement



Characteristics

A question mark.

A question mark (followed by a blank) is automatically displayed as a standard prompt when executing an INPUT statement, even though the statement does not include a "prompt string".

Self Prompting

By inserting a "prompt string" in an INPUT statement the user can display the message he requires (self prompting) in order to remember the value(s) to be entered.

Suppression of the standard prompt.

The user can eliminate the standard prompt (?) by writing a comma (instead of a semi-colon) after the prompt.

Suppression of the CR echo.

The echo of CR on the screen can be suppressed writing a semi-colon (;) after the INPUT keyword.

To enter a list of data.

The INPUT statement allows entry of one or more numeric or string data items from the keyboard. The type of data entered from keyboard must match the type of the variable to which the data is to be assigned. i.e. numeric variables require numeric constants, as data, (conversion from one numeric type to another is allowed, for example, a double precision constant can be associated with an integer varibale) and string variables require quoted or unquoted strings as data.
A string must be quoted if it contains commas or initial or final blanks. Numeric items may be input into string variables using an INPUT statement. If a number is introduced into a variable string, this is interpreted as the sequence of matching ASCII characters.

If too many or too few data items are introduced in reply to INPUT, or an incorrect type of data is entered, "Redo from Start?" will be displayed. Input values will not be assigned until an acceptable reply is given. When the statement INPUT; A is excuted the message "Redo from Start?" will be not displayed and the value entered will be filled with blanks.

Example 1

```
10 INPUT X
20 PRINT X "SQUARED IS" X^2
30 END
OK
RUN
? 5
5 SQUARED IS 25
OK
```

When statement 10 is executed the standard prompt is displayed (?) indicating that the program is waiting for data.

In this case, no prompt string clause is used in the INPUT statement (see statement 10).

Example 2

```
10 PI = 3.1415
20 INPUT "Radius";R
30 A = PI*R^2
40 PRINT "Area";A
50 GOTO 20
OK
RUN
Radius? 7.4
Area 172.029
Radius?
etc.
```

Execution of statement 20 causes the user prompt (Radius) to be displayed in front of the standard prompt (?).

Example 3

```
10 INPUT "Date", D$
20 PRINT D$
OK
RUN
Date30/Oct/69
30/Oct/69
OK
```

The standard prompt (?) is suppressed because in statement 10 the user prompt (Date) is followed by a comma (,).

Example 4

```
10 INPUT; "Date";D$
20 PRINT " J.C."
OK
RUN
Date? 30/Oct/69 J.C
```

The echo on the screen of the carriage return is suppressed by inserting a semi-colon (;) immediately after the keyword INPUT (see statement 10).

The next PRINT/INPUT operation (see statement 20) will be executed from the next screen position.

Example 5

```
10 INPUT A,B$,C(3)
20 PRINT A;B$;C(3)
30 GOTO 10
OK
RUN
? 1.2,ABC,4
   1.2 ABC 4
? ABD,1,3,5
? Redo from start
? 1.3,ABD,5
   1.3 ABD 5
?^C
Break in 10
OK
```

When statement 10 is executed, three data-items must be entered. The first must be numeric (1.2), the second a string (ABC) (this need not be quoted) and the third numeric (4).

They will be assigned to variables A, B$ and C (3) respectively.

When statement 10 is executed for the second time and an incorrect data item is entered, i.e. a string instead of a number, the system displays:

? Redo from start

and the data must be re-entered.

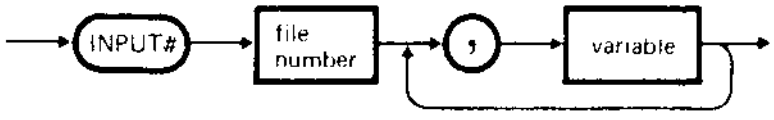
/CTRL/ /C/ must be pressed to interrupt execution of the program.

To resume execution, /CONT/ must be pressed.

INPUT#

Reads data from a sequential file and assigns them to program variables.

PROGRAM/IMMEDIATE Statement



where:

file number — is a numeric expression whose rounded value specifies the file number.

variable — is the name of the variable which will contain a data item from the file.

Characteristics — Unlike the INPUT statement, the INPUT# statement does not display a prompt (?) when executed.

Note — When an INPUT# statement is executed, data is read by the input sequentially from the specified file; that is, when the file is first opened, the file pointer is set to the beginning of the file.
Each time a data item is entered, the pointer moves to the next data item.
To read back the file from the beginning, it must be closed and then re-opened.

To avoid errors during the reading of the file, the type (numeric or string) of each data item in the file must be known. Data items in the file must be separated by appropriate delimiters.

Numeric data items may be input into string variables.
If a number is input into a string, the VAL function must be used to obtain the numeric value, to avoid type mismatch errors.

When a data item is assigned to a numeric variable, BASIC ignores any leading spaces, carriage returns and line feeds.

The first character encountered that is not a space, carriage return or line feed is assumed to be the start of a number.
The number ends when a space, comma, carriage return or line feed is found.

During assignment of numeric constants to a variable, there may be a conversion, if the variable is of a different type from the constant (as may happen with the LET or INPUT or READ statements).

When BASIC is inputting to a string variable, any leading spaces, carriage returns and line feeds will be ignored.
The first character found that is not a space, carriage return or line feed is assumed to be the first character of a string.

If this first character is a quotation mark ("), the string consists of all the charcters read between the first quotation mark and the second.

The quotation marks are not part of the string (this means that a quoted string on disk may not contain a quotation mark as a character).

If the first character is not a quotation mark, the string is not quoted and ends when a comma is found or a carrigae return or line feed (after 255 characters have been read).
For example if the following data is on disk:

```
SUBROUTINES, SUBPROGRAMS "HOW TO CALL THEM?"
```

the statement:

```
INPUT # 1,R$,S$,T$
```

will assign the values as follows:

```
R$ = SUBROUTINES
S$ = SUBPROGRAMS "HOW TO CALL THEM?"
```

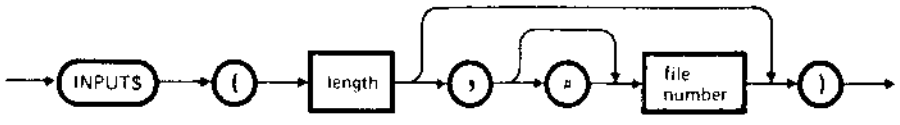
If the following data are on disk:

```
SUBROUTINES, SUBPROGRAMS "HOW TO CALL THEM?"
```

the same statement will assign the following values:

```
R$ = SUBROUTINES
R$ = SUBPROGRAMS
T$ = HOW TO CALL THEM?
```

This function returns a string of specified length from the keyboard or read from a disk file. No characters will be echoed and all control characters are passed through to the program except for /CTRL/ /C/ which interrupts program execution.



where:

length — is a numeric expression rounded to the nearest integer. It specifies the length of the string.

file number — is the number of the file.

Example 1

```
10 OPEN"I",1,"DATA"
20 IF EOF (1) THEN 50
30 PRINT HEX$(ASC(INPUT$(1,#1)));
40 GOTO 20
50 PRINT
60 END
```

This program lists the contents of a sequential file in hexadecimal characters.

Example 2

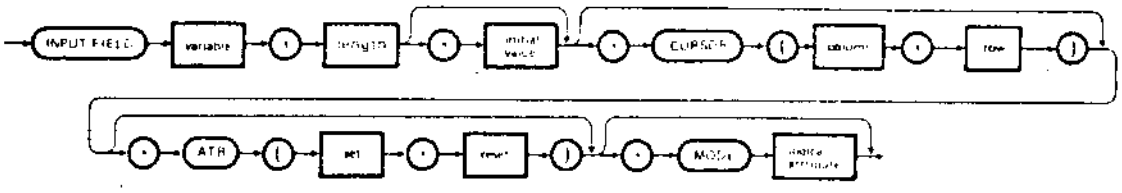
```
110 X$=INPUT$(1)
120 IF X$="S" THEN END
```

S must be entered to end the program or impede any character to continue.

INPUT FIELD

Allows a field to be entered from terminal during program execution.

PROGRAM/IMMEDIATE Statement



where:

variable	numeric or string variable (including subscripted variables) to which the result of the input operations will be assigned.
length	maximum length of the variable parameter
initial value	if specified, it will be displayed before accepting any input operation. If omitted, the field keeps the previous value.
column, row	where specified determine the position (column, row) of the cursor before the beginning of the field on the screen. column is an integer from 1 to 80 row is an integer from 1 to 24.

set, reset

where specified, they set and reset the visual attribute starting from the current cursor position for the entire length of the field specified by the parameter length.

set and reset are integers from 0 to 255with the following meaning:

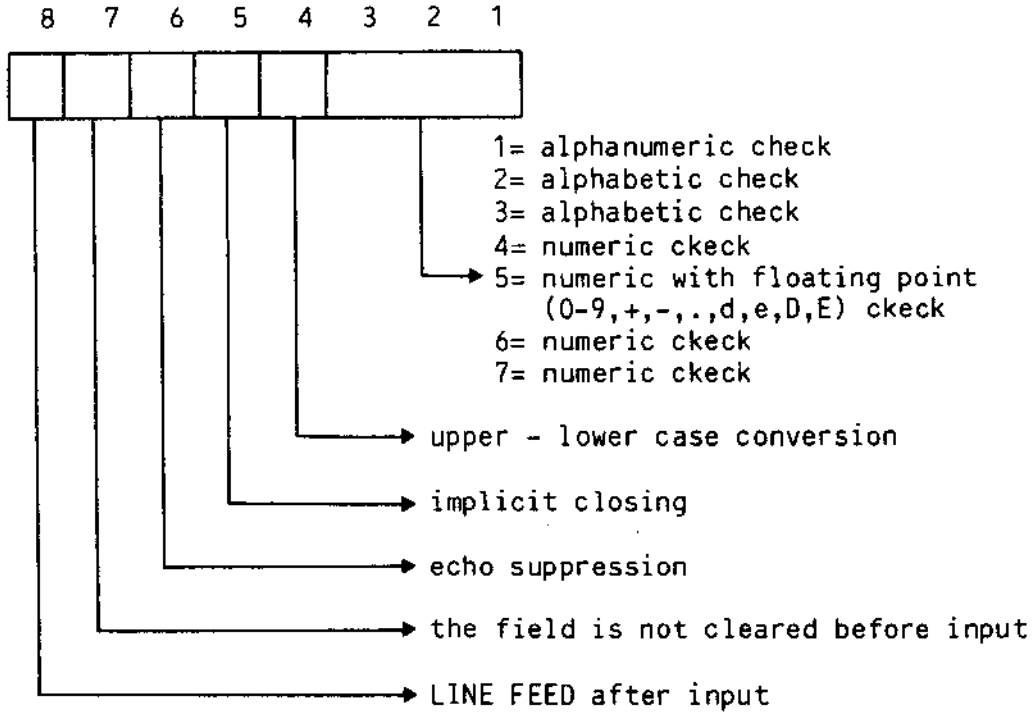
```
  1= upperscore
  2= underscore
  4= left-hand side
  8= right-hand side
 16= blinking
 32= high-light
 64= reverse
128= reserved for colour
  0= no effect.
```

32 and 64 together gives hide.

Any combination of the above is allowed.

logical attribute

Where specified, determines the check to be made during input.
logical attribute is an integer between 0 and 255.
It has the following meaning.



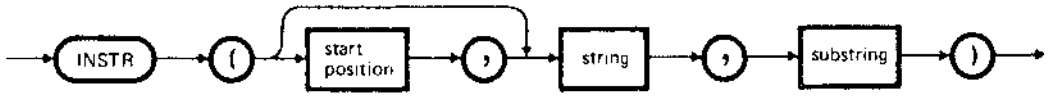
Any combination of the above is allowed.

Example

```
   ...
20 PRINT CURSOR(1,5) "ENTER ACCOUNT NUMBER"
30 PRINT CURSOR(1,7) "ENTER CUSTOMER NAME"
40 PRINT CURSOR(1,10) "ENTER AMOUNT TO BE WITHDRAWN"
50 PRINT CURSOR(1,13) "BALANCE"
60 PRINT CURSOR(1,18) "OPERATION OK?"
70 INPUT FIELD A$,6,CURSOR(30,5),ATR(32,0),MODE 1
80 INPUT FIELD B$,30,CURSOR(30,7),ATR(2,0),MODE 2
90 INPUT FIELD C$,7,CURSOR(30,10),ATR(64,0),MODE 5
100 PRINT ATR(16,0) CURSOR(30,13) D$
110 INPUT FIELD E$,2,"YES",CURSOR(30,18),MODE 2
120 IF E$="NO" THEN 70
```

Searches for the first substring in a string and returns the position in which it has been found.

PROGRAM/IMMEDIATE Statement



where:

start position — is a numeric expression rounded up to the nearest integer which specifies where the search must start. Its value must be between 1 and 255. If omitted,1 is assumed by defualt.

string — is a string expression or string variable or string constant or array element, whose value matches the string to be found.

substring — is a string constant or string variable or string expression or array element, which is being searced for.

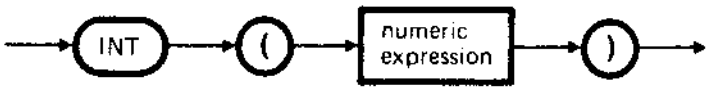
Characteristics — If start position > LEN (string), the value returned by this function is 0.
In the case of a null string (''''), the value returned is 0.
If a substring is not found, the function returns the value 0.
If the substring is a null string and start position is specified, the function value is equal to that of start position.

If the substring is a null string and start position has not been specified, the function value is 1.

Example

```
10 X$ = "ABCDEB"
20 Y$ = "B"
30 PRINT INSTR(X$,Y$);INSTR(4,X$,Y$)
RUN
 2 6
OK
```

This function provides the integer which is larger than, less than or equal to the argument.



Note

Note the difference between INT and FIX. With negative values, the returned value for INT is always less than or equal to the argument, whilst for FIX it is always greater than or equal to the argument.

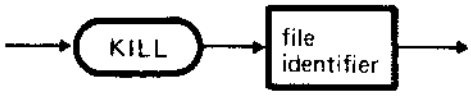
Examples

```
PRINT INT(99.89)
  99
OK
PRINT INT(-12.11)
-13
OK
```

KILL

Deletes a program or a data file stored on disk.

PROGRAM/IMMEDIATE Command



where:

file identifier — is a string constant or string variable or string expression or array element. It specifies the file to be deleted.

Characteristics — The KILL command can be used for all types of disk files, be program files, random files, keyed and sequential files. A file must be closed before entering KILL.

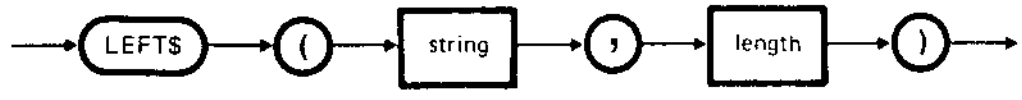
Example

```
KILL "Business.B"
```

The Business.B file is deleted.

LEFT$

This function returns a substring which is described from a given string and is comprised of the leftmost characters of a specified length.



where

string — is the string expression whose value matches the string from which the substring is to be extracted.

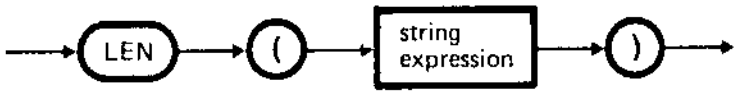
length — is a numeric expression rounded up to the nearest integer (from 0 to 255) whose value represents the length of the string required.

Characteristics — if the value of the parameter length = 0, the null string is returned.
If length>=LEN (string), the entire string is returned.

Example

```
10 A$ = "BASIC LANGUAGE"
20 B$ = LEFT$(A$,5)
30 PRINT B$
RUN
BASIC
OK
```

This function calculates the length of a specified string by counting all the characters (printable or otherwise) and the blanks.

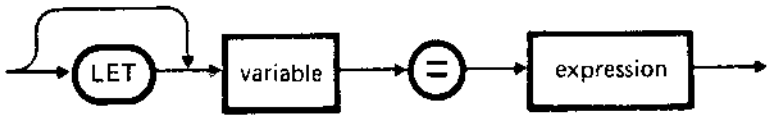


Example

```
10 X$ = "PORTLAND, OREGON"
20 PRINT LEN(X$)
RUN
 16
OK
```

Assigns the value of an expression to a variable.

PROGRAM/IMMEDIATE Statement



Characteristics

The LET keyword is optional. To assign an expression to a variable name, the = sign is used.
Simultaneous assignments are not allowed.

Example 1

```
LET K = 1.5
```

The value 1.5 is assigned to numeric variable K.

Example 2

```
LET X = K + 2
```

The value of the numeric expression K +2 is assigned to the numeric variable X

Example 3

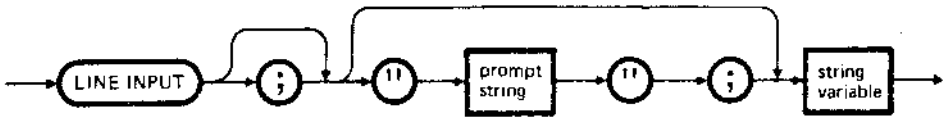
```
A$ = "ABC"
```

The value of the string constant "ABC" is assigned to the string variable A$.

LINE INPUT

Allows input of an entire line up to a carriage return/line feed and assigns it to a string variable without the use of delimiters (255 characters is the maximum length of a line).

PROGRAM Statement



where:

prompt string — is a literal string that is displayed before the input is accepted.

Characteristics — The standard prompt (?) does not appear when executing the LINE INPUT statement. If the question mark is to be displayed, it must be placed at the end of the "prompt string" clause.
The echo on the screen may be eliminated entering a semi-colon (;) immediately after LINE INPUT.

Example 1

```
10 LINE INPUT "Name? ";N$
20 PRINT "JONES"
OK
RUN
Name? LINDA
JONES
OK
```

The prompt string (Name?) is displayed before data is entered.

All the characters entered from the end of the prompt to CR are assigned to the string variable (N$).

Example 2

```
10 LINE INPUT;"Name? ";N$
20 PRINT " JONES"
OK
RUN
Name? LINDA JONES
OK
```

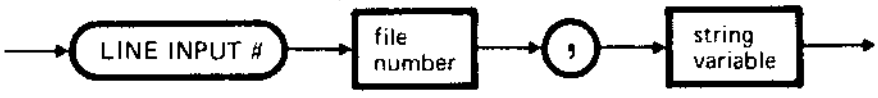
The echo on the screen of CR may be eliminated entering a semi-colon (;) after LINE INPUT (see statement 10).

The next PRINT/INPUT operation (see statement 20) will be executed starting from the next screen position.

LINE INPUT

Reads an entire line (up to carriage return) from a sequential file and assigns it to a string variable.

PROGRAM/IMMEDIATE Statement



where:

file number — is a numeric expression whose value specifies the number of the file.

string variable — is the name of the variable to which the line will be assigned.

Characteristics — If a LINE INPUT # statement is executed, a line of string data is read into the specified string variable.

LINE INPUT # reads all characters in the file up to:

- a carriage return or
- a carriage return/line feed or
- the end of the file
- the 255th characters (this is included in the string).

when a data item is assigned to a string variable, BASIC ignores any leading blanks, carriage

return/line feed characters.
When the first character that is not a leading blank, carriage return or line feed character is encountered, BASIC assumes that this is the first character of a string. For example, if the following data are on the disk:

SUBROUTINES, SUBPROGRAMS "HOW TO CALL THEM?"

the following statements:

```
INPUT # 1,R$,S$,T$
PRINT R$, S$
```

will assign and display the values as follows:

R$ = SUBROUTINES
S$ = SUBPROGRAMS "HOW TO CALL THEM?"

If the following data were recorded on disk:

SUBROUTINES, SUBPROGRAMS, "HOW TO CALL THEM?"

the LINE INPUT #1, X$:PRINT X$ statements will assign and display the following values:

X$ = SUBROUTINE, SUBPROGRAMS, "HOW TO CALL THEM?"

If leading characters or other delimiters are encountered such as quotation marks, commas, blanks and so on, they are all included in the string.

If data are to be read without following the normal recordings as regards leading and end characters, the LINE INPUT # statement must be used.

Note

LINE INPUT # reads all the characters in the sequential file up to a carriage return. If then skips the carriage return/line feed sequence and the next LINE INPUT # reads all the characters up to the next carriage return. (If a line feed/carriage return sequence is encountered, it is considered as part of the string variable).

Example

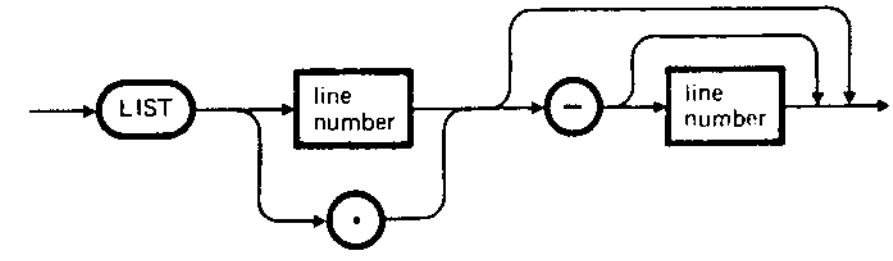
```
10 INPUT "PROGRAM IDENTIFIER";P$
20 OPEN "I",1,P$
30 K%=0
40 IF EOF(1) THEN 80
50 K%=K%+1
60 LINE INPUT#1,A$
70 GOTO 40
80 PRINT P$ " IS" K% "LINES LONG"
90 CLOSE
```

```
100 GOTO 10
110 END
OK
RUN
PROGRAM IDENTIFIER? P1
P1 IS 350 LINES LONG
PROGRAM IDENTIFIER? P2
P2 IS 1520 LINES LONG
PROGRAM IDENTIFIER?^C
Break in 10
OK
```

This program counts the number of lines in an ASCII format program file.
Each line ends with a carriage return/line feed, thus LINE INPUT # at line 60 reads one entire line at a time into the dummy variable A$. Variable K% counts the line of the program.

Lists on screen all or part of the program held in the memory.

IMMEDIATE Command



Example 1

LIST
The program held in the memory is listed.

Example 2

LIST 500
Line 500 of the program in the memory is listed.

Example 3

LIST 150-
Program lines from 150 to the end of the program are listed.

Example 4

LIST -1000
The program is listed from the beginning to line 1000.

Example 5

LIST 150 - 1000
All the lines between 150 and 1000 are listed.

Example 6

LIST 100-.
All the lines from 100 including the present line are listed.

Example 7

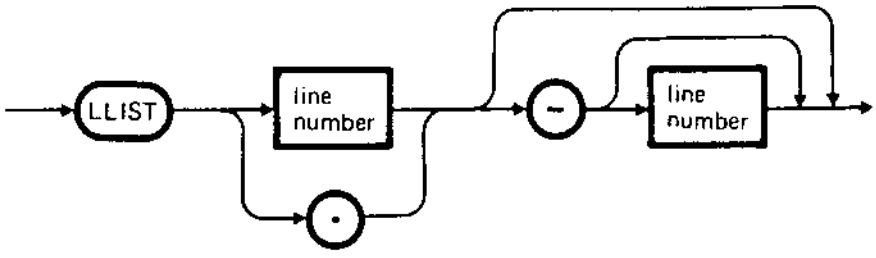
```
LIST.-300
```

All the lines from the present line to line 300 included are listed.

LLIST

Prints all or part of a program held in the memory.

IMMEDIATE Command



Characteristics

The LLIST command has the same features as the LIST command.

Examples

See the examples of the LIST command.

LOAD

Transfers a program from disk to memory.

PROGRAM/IMMEDIATE Command

where:

file identifier can be either a constant or a variable string. It specifies the program file which must be transferred from disk to memory. The value of the file identifier must be the same as that used when the program was stored on disk.

R specifies that all data files opened by the previous program are left open and that the new program, once transferred from disk to memory, is also executed. If R option is not specified, the LOAD command closes all the data files which were opened by the previous program and cancels all program variables and lines resident in memory before the start of program execution.

Example 1

```
LOAD "RECTANGLE1"
```

The RECTANGLE1 program is transferred from disk to memory.

Example 2

```
LOAD B$
```

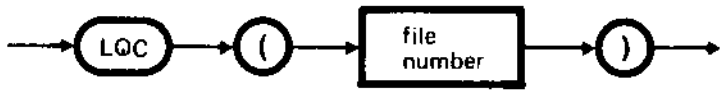
The program, identified by the contents of the B$ variable, is transferred from disk to memory.

Example 3

```
LOAD "ACCOUNT",R
```

The ACCOUNT program is transferred from disk to memory and executed; all files opened by the program are left open.

For random files, the LOC function provides the record number previously read using a GET statement or previously stored using a PUT statement.
For sequential files, LOC provides the buffer numbers that are passed from the BASIC interpreter to the File System.



where:

file number — is a numeric expression that is rounded up to the nearest integer. It is the file number.

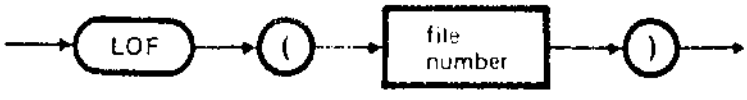
Characteristics — For random files, if no I/O operation has been carried out, LOC provides the value 0. For sequential files, if no I/O operation has been carried out, LOC provides the value 1 as input and the value of 0 as output.

Example

```
200 IF LOC(2)>30 THEN STOP
```

LOF(LENGTH OF FILE)

This function provides the length of a sequential file.

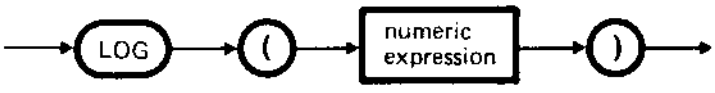


where:

file number — is a numeric expression that is rounded up to the nearest integer. It is the file number.

Characteristics — The length of the file is provided in terms of numbers of blocks (each of 512 bytes).

This function calculates the natural logarithm.
Calculation is carried out in double precision.



where:

numeric expression — must be positive.

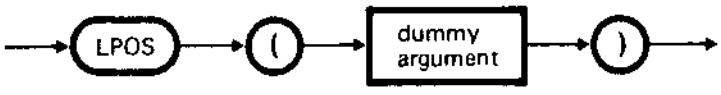
Characteristics — On the basis of the equasion $\log_a x = \log_e x / \log_e a$ it is possible to calculate the logarithm in base 10 (or in any other base).

Note — The numeric expression can be of any type the result is always in double precision. For a correct and precise result, it is advisable to use a numeric expression in double precision as input.

Example

```
PRINT LOG(45/7)
 1.86075230892586
OK
```

This function provides the current print-head position (within the limits of the printer's line buffer).



where:

dummy argument — is any string or numeric expression. Its result does not depend on the given argument.

Characteristics — The position provided by the LPOS function does not necessarily correspond to the actual physical location of the print head.

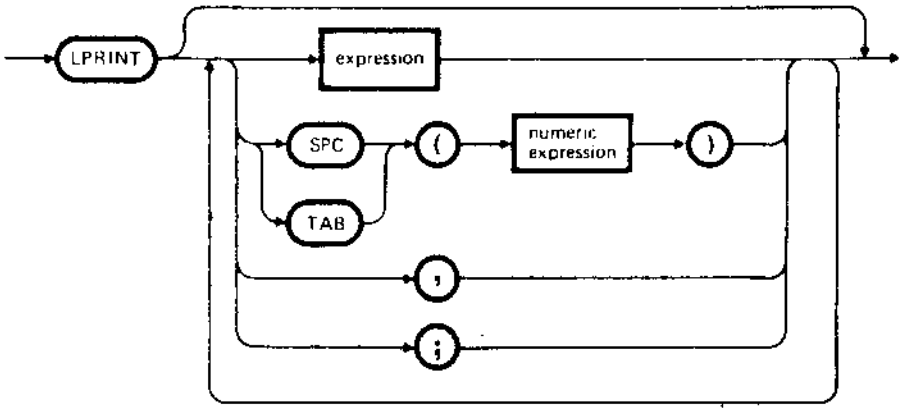
Example

```
100 IF LPOS(X)>60 THEN LPRINT CHR$(13)
```

LPRINT

Allows data to be printed in a standard format.

PROGRAM/IMMEDIATE Statement

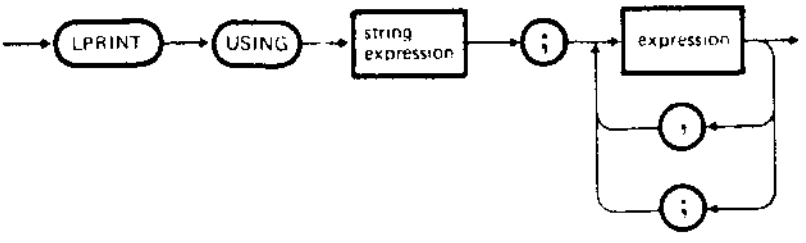


Characteristics

LPRINT has the same characteristics as PRINT. The only difference is that, in this case, output is on a printer.

LPRINT USING

Allows data to be printed according to a user-defined format.



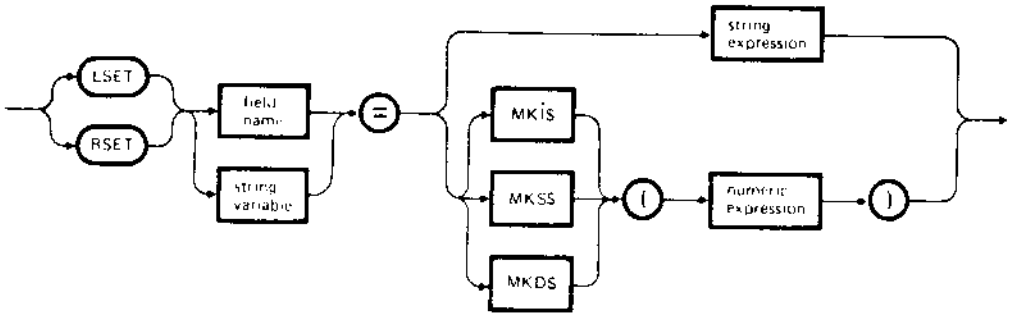
Characteristics

LPRINT USING has the same characteristics as the PRINT USING statement.
The only difference is that, in this case, output is on a printer.

LSET stores a string value aligned to the left of a random access buffer field or it aligns a string value on the left in a string variable.

RSET stores a string value aligned to the right of a random access buffer field, or it aligns a string value on the right in a string variable.

PROGRAM/IMMEDIATE Statements



where:

field name	is a string variable specifying the name of a random buffer field.
string variable	is the name of an ordinary string variable.
MKI$/MKS$/MKD$	are the functions which convert an integer (MKI$), or a value in single precision (MKS$), or in double precision (MKD$), into a string value.

string expression — is the string to be aligned either to the left or right in a given field.

numeric expression — is the numeric value to be converted into a string and aligned either to the left or right of a given field.

Example 1

```
10 OPEN "R",#1,"MYFILE/MYPASS",20
20 FIELD#1,10 AS N1$,10 AS N2$
30 LSET N1$="CHARLES"
40 LSET N2$="JAMES"
    .
    .
    .
100 RSET N1$="CHARLES"
110 RSET N2$="JAMES"
    .
    .
    .
200 LSET N1$="CHARLES THOMSON"
    .
    .
    .
```

statements 30 and 40 assign data on the buffer #1 as follows:

N1$

CHARLES

N2$

JAMES

Statements 100 and 110 assign data in the buffer as follows:

N1$

   CHARLES

N2$

   JAMES

Statement 200 assigns data in the buffer as follows:

N1$

CHARLES TH

If a string is too long and cannot be contained in

the specified buffer field, it is truncated on the right, regardless of whether the LSET or RSET statement has been used.

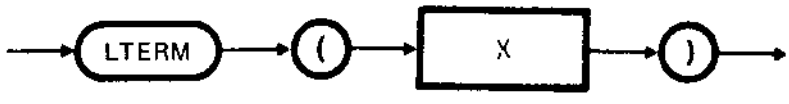
Example 2

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

Statements LSET and RSET can also be used with a variable not associated to the statement FIELD# for aligning to the right or left of a string in a given field. This is a useful formatting technique for printing operations.

In the example on the left the RSET statement aligns the N$ string to the right of a 20 character field.

This function returns an integer value indicating the last end of input key, or the number of characters entered during the last input operation.



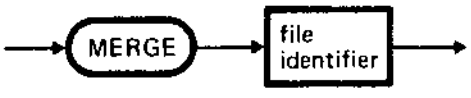
Characteristics

If the value 0 is assigned to the parameter X, the LTERM function returns an integer which indicates the end of input key entered during the last input operation (see Appendix B for a list of end of input keys). If a value other than zero is assigned to X, LTERM still returns an integer value, but this time it indicates the number of characters keyed during the last input.

MERGE

Executes a MERGE operation between the program in memory and the program stored on disk.

PROGRAM/IMMEDIATE Statement



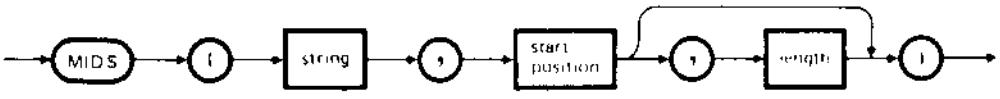
where:

file identifier — can be a constant or a string variable specifying a program file in ASCII format.

Characteristics — MERGE closes all data files that were opened by the program resident in memory, which are not yet closed. It makes the numeric variables zero and initialises string variables with the string value null ("").

Example — MERGE "Numbers"

This function extracts a sub-string from an assigned string, starting from a given character position.



where:

string	is a string expression whose value corresponds to the string from which the sub-string must be taken.
start position	is a numeric expression rounded up to the nearest integer, whose value (>=1 and< = of the string length) specifies the position of the first character from where the sub-string must be extracted.
length	is a numeric expression rounded up to the nearest integer, whose value (from 0 to 255) represents the length of the sub-string to be extracted. If omitted all the characters are extracted from the location defined by start position at the end of the string. If length equals 0 the function provides the string null.
Characteristics	If the number of characters on the right of start position is less from that specified by length, the function extracts all the characters from the location defined by start position at the end of the assigned string. If start position > LEN (string) the function provides the string null.

Note

See LEFT$ and RIGHT$.

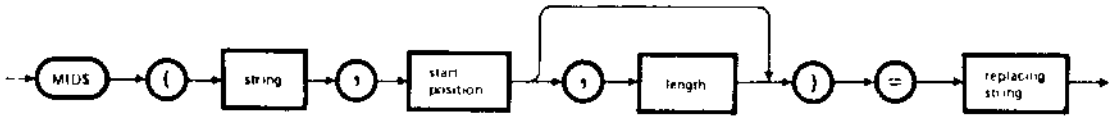
Example

```
10 A$="GOOD "
20 B$="MORNING EVENING AFTERNOON"
30 PRINT A$;MID$(B$,9,7)
OK
RUN
GOOD EVENING
OK
```

MID$

Replaces all or part of a data string with another string.

PROGRAM/IMMEDIATE Statement



where:

string	is a string expression whose value is the string, part of which is to be replaced.
start position	this is a numeric expression rounded to the nearest integer whose values must be >=1 and <= of the length of the string. This value specifies the character position of the beginning of the replacement.
length	this is a numeric expression rounded to the nearest integer whose value (from 0 to 255) represents the length of the replacing string parameter. If length is omitted, all the characters from start position to the end of replacing string are replaced. However, regardless of whether length is specified or not, replacement of characters ends with the last character of the original string.
replacing string	this is a string expression which replaces the characters in the original string, beginning at the start position. If it assumes the string value null, the MID$ statement has no effect.

Example

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

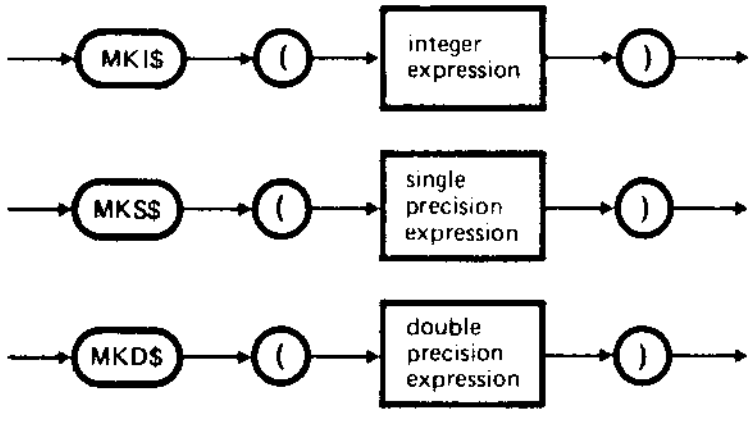
MKI$/MKS$/MKD$

These functions change numeric values into string values.

The MKI$ function converts an integer into a 2-character string.

The MKS$ function converts a single precision value to a 4-character string.

The MKD$ function converts a double precision value into a 8-character string.



Note

A numeric value in a buffer of a random or keyed file must be converted into a string with the LSET or RSET statement.

Example 1

```
30 LSET D$=MKI$(I%)
```

Field name D$ should now contain a two byte representation of the integer I%.

Example 2

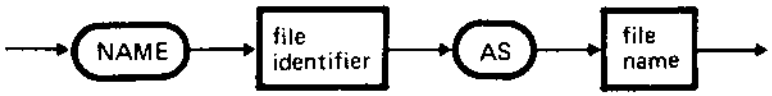
```
100 STR4C$=MKS$(SPV)
```

A "make" function (MKI$, MKS$, MKD$) must be strictly used with the LSET and RSET statements. In this case, SPV is the name of a single precision variable which is converted into a 4-character string and assigned to the STR4C$ variable.

NAME

Changes the name of a disk file.

PROGRAM/IMMEDIATE Statement



where:

file identifier can be either a string constant or a string variable which specifies the program or data file whose name is to be changed.

file name can be either a string constant or a string variable which specifies the name of the new file.

Note After execution of the NAME statement, the file exists on the same disk, in the same area with its new name.

Example

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

File name ACCTS is changed into LEDGER

NEW

Deletes the current program and variables allowing entry of a new program.

IMMEDIATE Command

Characteristics

NEW need not be entered before loading a program from disk with the LOAD and RUN commands (these commands automatically clear the memory).

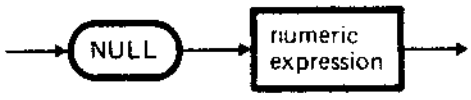
Example

```
NEW
```

The program currently in memory is deleted together with its variables.
A new program can now be entered from keyboard.

Sets the number of nulls to be printed at the end of each line.

PROGRAM/IMMEDIATE Statement

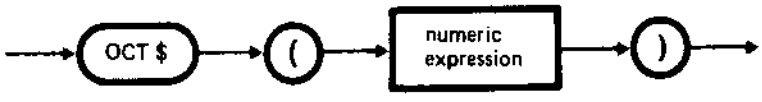


Example

```
NULL 2
```

2 nulls will be printed at the end of each line.

This function returns a string that represents the octal value of a decimal argument.



where:

numeric expression	is rounded to the nearest integer
Note	See the HEX$ function for hexadecimal conversion.
Example	PRINT OCT$(24) 30 OK

ON ERROR GOTO

Enables error handling and specifies the first line of the error handling routine.

PROGRAM Statement

where:

line number — is the first line of the error handling routine. It must be greater than 0 or less than or equal to 65529.

Characteristics — If the line number after an ERROR GO TO does not exist, the following message is displayed:

Undefined line number

If the ON ERROR GOTO 0 statement is executed, non standard error handling is disabled. Upon subsequent errors, the standard error message is displayed and execution is interrupted.

If the ON ERROR GOTO 0 statement is inserted in an error trap routine, the program displays the standard error message which caused the trap and stops.

It is recommended that all error handling routines execute an ON ERROR GOTO 0 statement if an error is encountered for which there is no recovery action.

Two error handling routines cannot be active at the same time. Therefore, if an error has occurred during execution of a non standard error handling routine, the program displays an error message and stops. Control cannot be passed to the error handling routine when an error occurs within the same routine.

Note

Overflow and Division by Zero errors cannot be trapped.

Example

```
 .
 .
 .
110 ON ERROR GOTO 400
120 INPUT "WHAT IS YOUR BET";B
130 IF B>5000THEN ERROR 210
 .
 .
 .
400 IF ERR-210 THEN PRINT "HOUSE LIMIT IS $5000"
410 IF ERL-130 THEN RESUME 120
420 ON ERROR GOTO 0
 .
 .
 .
```

If a value of B greater than 5000 is entered, the following message is displayed:

```
HOUSE LIMIT IS $5000
```

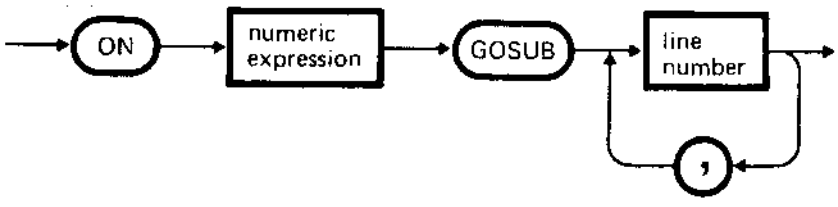
and execution resumes at line 120. If any other errors are encountered, statement 420 standard error message is displayed.

ON...GOSUB/RETURN

The ON...GOSUB statement calls one of several subroutines.

The RETURN statement passes control to the statement following the most recent ON...GOSUB that has been executed.

PROGRAM Statement



---

where:

numeric expression — specifies the subroutine to be called. It must be between 0 and 255.

- if the value is 1, the subroutine at the first line number in the list will be called
- if the value is 2, the subroutine at the second line number in the list will be called and so on.
- if the value is not an integer, it is rounded to the nearest integer

- if the value is zero, or greater than the number of elements in the list, (but less than or equal to 255), the program continues with the next executable statement after ON...GOSUB.

line number — each line in the list must be the first line number of a subroutine.

Example

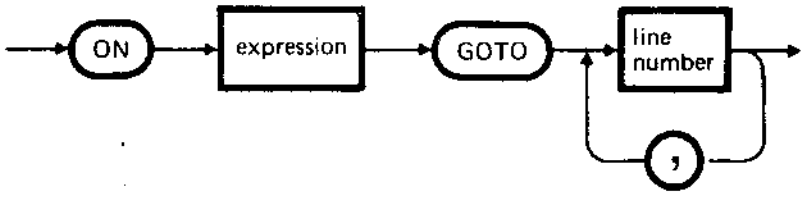
```
10 INPUT "Enter 1,2 or 3";K%
20 ON K% GOSUB 40,50,60
30 END
40 PRINT "SUB1":RETURN
50 PRINT "SUB2":RETURN
60 PRINT "SUB3":RETURN
OK
RUN
Enter 1,2 or 3? 2
SUB2
OK
```

If the user enters 1, the program displays SUB1, if 2 is entered, SUB2 is displayed, if 3 is entered, SUB3 is displayed.
In each of these cases, a RETURN statement will transfer control to END.
If an integer between 0 and 255 is entered, other than 1, 2 or 3, the program will display nothing.

ON...GOTO

Passes control to one of several specified lines after GOTO according to the value of an expression specified after ON.

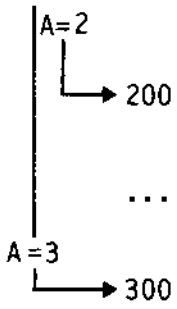
PROGRAM/IMMEDIATE Statement



Characteristics

Assuming that the program has the following structure:

```
20 INPUT A
30 ON A GOTO 100,200,300
40
...
90
100
...
```



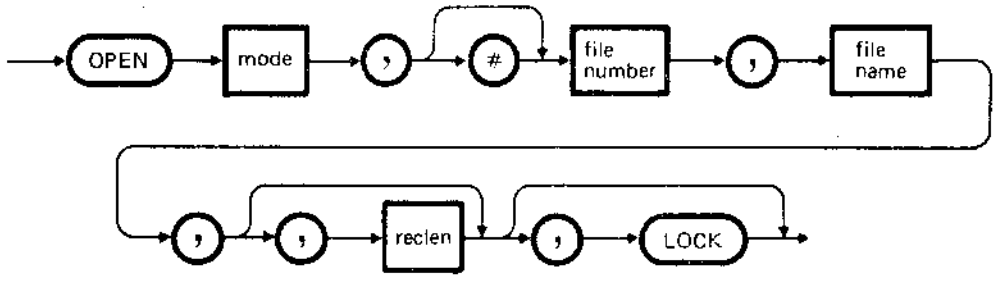
The value of A determines to which line number in the list control will be passed. If, for example, the value is 3, control will be passed to the third line number in the list (if the value of A is non integer, it is rounded to the nearest integer).

If the value of the expression after ON, is equal to 0 or greater than the number of elements in the list, (but less than or equal to 255), control is passed to the first executable statement that follows.

OPEN

Opens a data file.

PROGRAM/IMMEDIATE Statement



where:

mode — is either a string constant or a string variable or an element of an array whose first character may be:

- "A", i.e. Append: sequential output after the last data item in a sequential file. Data in the file (if any) are not lost; new data will be added at the end of the file.
- "I" i.e. Input: sequential input starting from the beginning of the file, which is sequential.
- "O" i.e. Output: sequential output starting from the start of the file, which is sequential. Data in the file, if any, are lost.
- "R" i.e. Random: records may be input/output in any order. The file, in this case, is positional.
- "K" i.e. Keyed: records can be input/output/deleted in any order. In this case,

the file is keyed (with single or multiple key). If a sequential file is empty (i.e. it does not contain data) access mode A and access mode O are equivalent.

file number — numeric expression whose value (rounded to the nearest integer) must be between 1 and 15. The specified file number remains associated with the file for all the time this remains open and is used to specify that particular file in each I/O statement.

file name — this is either a string constant, a string variable, a string expression or an array element containing a maximum of 60 characters. It may specify:

- a new file (i.e. unknown to the system). In this case, the file is created (except for access mode "I" and "K").
- the file already exists. In this case, the file is only opened.

file identifer may indicate:

. a complete pathname

. the name of the file in the working directory

. a local name to which an global value is assigned before BASIC is started in a SHELL environment with the commands Shell: CONN, DISCON.

reclen — this is a numeric expression (rounded to the nearest integer) which, when specified, sets the record length of a random or keyed file. This parameter can be specified only for random or keyed files. Its default value is 256 bytes or the value of the parameter set by the user when BASIC is started. The length may be between 1 and 4096 bytes.

LOCK — prevents any other user accessing the file until this has been closed.

Note

In the case of keyed files, the CREATE statement must be used for file creation. The use of CREATE is recommended also for other types of file for a better control of disk space.

Example

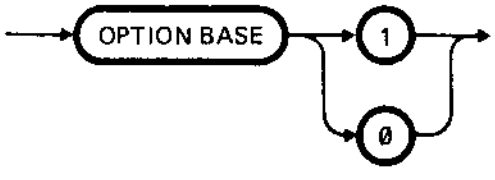
```
    .
    .
    .
50 OPEN "A",1,"EXAMPLE"
    .
    .
    .
160 OPEN "O",2,"TEST"
    .
    .
    .
270 OPEN "R",3,"F1",80
280 OPEN "R",4,"F2",20
    .
    .
    .
490 CLOSE 2
500 OPEN "I",5,"TEST"
    .
    .
    .
600 OPEN "R",2,FILES,RN
```

Statement 50 opens the sequential file EXAMPLE in Append access mode. File number 1 is associated with the file.
Statement 160 opens the sequential file TEST in Output access mode associating file number 2.
Statement 270 opens the random file F1, associating file number 3 and sets the record length to 80 bytes.
Statement 280 opens the random file F2, associating file number 4 and sets the record length at 40 bytes.
Statement 490 closes the TEST file.
Statement 500 re-opens the TEST file in Input access mode and associates file number 5 to the file.
Statement 600 opens a random file whose identifier is the contents of the string variable FILES. The record length is indicated by the contents of the numeric variable RN. The associated file number is 2, made available after statement 490.

OPTION BASE

Declares the lower bound for array subscripts.

PROGRAM/IMMEDIATE Statement



Characteristics

If the OPTION BASE statement is not executed, 0 is assumed by default as lower bound.
If executed, the OPTION BASE statement cannot be preceded by a DIM statement or by an array reference.
The OPTION BASE 1 statement is useful especially when converting programs written for other systems onto M30/M40. Some versions of BASIC number all arrays from 1.

Example

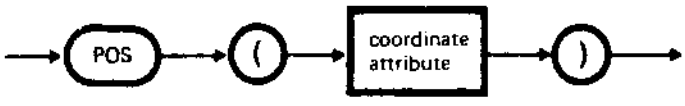
```
10 OPTION BASE 1
```

or

```
OPTION BASE 1
```

The lower bound of the subscripts is 1.

This function returns the position of the text cursor (row or column) in the current window.



where:

coordinate attribute — specifies whether the row or column position is to be returned.

0 : column position
0 : row position

Note — See the LPOS function.

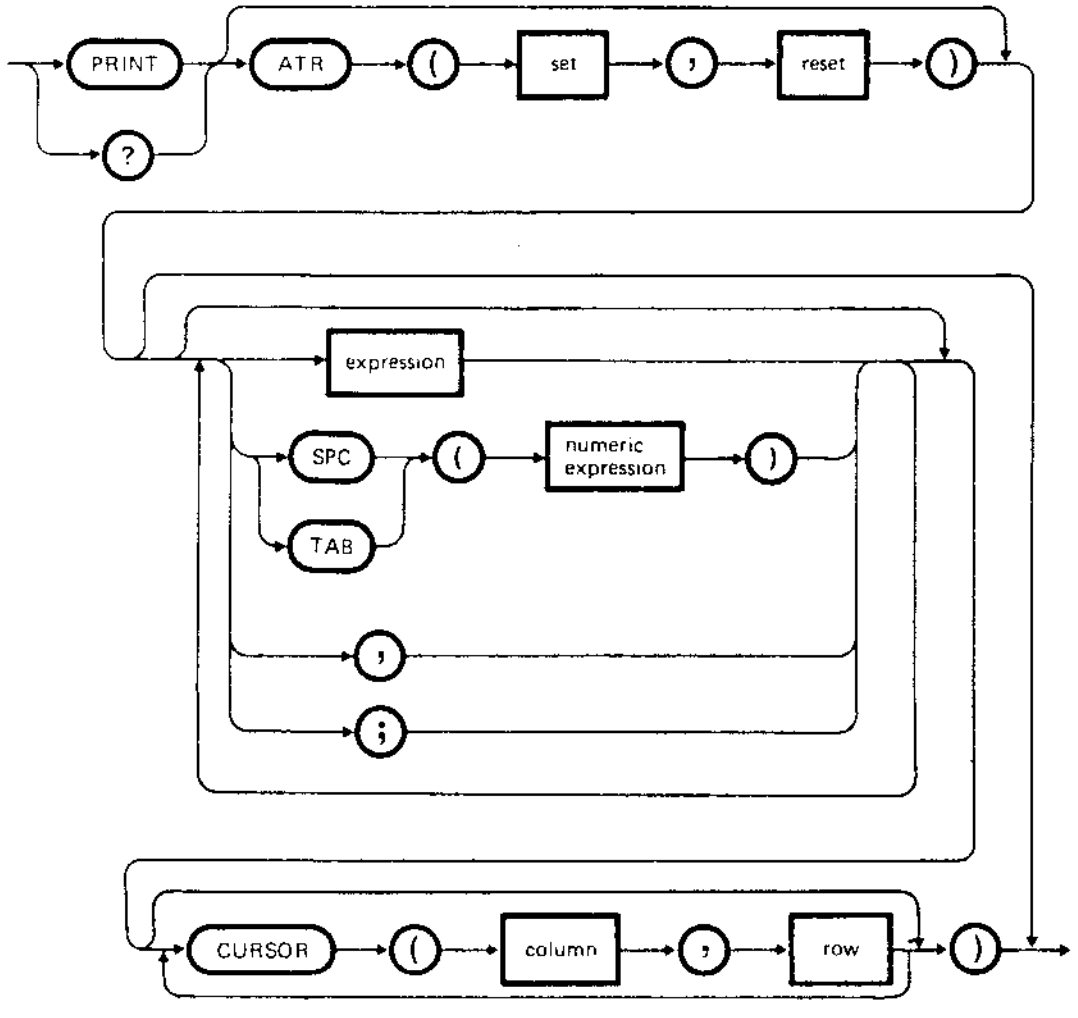
Example

```
IF POS(0)>60 THEN PRINT CHR$(13)
```

PRINT

Displays a list of data in a standard format.
A question mark (?) may be used instead of the PRINT keyword.

PROGRAM/IMMEDIATE Statement



where:

set, reset

where specified, set and reset the visual attributes for all the expression parameters indicated. set and reset are integers from 0 to 255 with the following meanings:

```
  0  no effect
  1  overscore
  2  underscore
  4  left-hand margin
  8  right-hand margin
 16  blinking
 32  high-light
 64  reverse
128  reserved for colour.
```

Any combination of the above is possible.

column, row

where specified, these set the cursor position from where the expression that follows is to be displayed.

Characteristics

If the PRINT statement does not end with a comma or a semi-colon a new line of output is generated.

For example:

```
10 PRINT 1
20 PRINT 2
OK
RUN
 1
 2
OK
```

If the PRINT statement does not contain any expressions, a line is skipped.

For example:

```
10 PRINT 1
20 PRINT
30 PRINT 2
OK
RUN
 1
 2
OK
```

Where the expressions in the output list are separated by a comma: each value is displayed left-justified in one of the "zones" into which each line is divided (each zone consists of 16 positions).

For example:

```
10 A$ = "For June..."
20 X = .353
30 PRINT "Results", A$, X
OK
RUN
Results      For June...      .353
OK
```

A space is inserted before each positive numeric value.
(in this case .353).

The number of print zones on each line depends on the maximum number of characters each can contain.

String values displayed are not between quotation marks.

If the list of expressions has many entries, two or more lines of output may be produced.

Each positive value is preceded by a space. If a PRINT statement contains one or more numeric expressions:

- each value which is printed or displayed is always followed by a space
- each positive value is preceded by a space
- each negative value is preceded by a minus sign
- each single precision value that can be represented with 6 or fewer digits in the fixed point format as accurately as it can be represented in floating point (or exponential format) is output using the fixed format

. each double precision value that can be represented with 15 or fewer digits in the fixed format as accurately as it can be represented in the floating format (or exponential format) is output using the fixed point format. For example:

```
PRINT 10^-6
 .000001
OK
PRINT 10^-7
 .000001
OK
PRINT 1D-15, 1D-16
 .0000000000000001       1D-16
OK
```

The second value is displayed left-justified in the third zone (as the first value overflows into the second print zone).
If the last expression of the list is followed by a comma, a subsequent PRINT or INPUT Operation, will cause the next character or digit to be displayed on the same line (at the start of the next print zone), if there is enough room. (Otherwise, it will be displayed on the next line.

For example:

```
10 A$ = "For July..."
20 X = .491
30 PRINT "Results", A$
40 PRINT X
OK
RUN
Results      For July...      .491
```

If the last expression of the list is followed by a semi-colon, the next PRINT or INPUT operation will display the next characters or digit on the same line at the cursor position, if there is enough space. Otherwise, it will be displayed on the next line.

For example:

```
10 INPUT X
20 PRINT X "SQUARED IS" X^2 "AND";
30 PRINT X "CUBED IS" X^3
40 PRINT
50 GOTO 10
OK
RUN
? 9
```

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

? 21
 21 SQUARED IS 441 AND 21 CUBED IS 9261

?
```

If commas are used consecutively, the effect of each comma is to position the cursor at the start of the next zone.

Using commas in this way, data can be displayed widely spaced.

For example:

```
PRINT "M",, "N"
M             N
OK
```

If semi-colons or blanks are used instead of commas to separate expressions in the list, the values displayed are closer together. The exact spacing depends on the number of digits or characters in each value. The use of semi-colons in this way allows display of more values on each line.

If more than one space or semi-colon is inserted between two expressions, this has the same effect as one space or one semi-colon.

For example:

```
LIST
10 A1 = 1000
20 A2 = 2000
30 A3 = 3000
40 A4 = 4000
50 A5 = 5000
60 A6 = 6000
70 A7 = -7000
80 PRINT A1;A2;A3;A4;;A5 A6 A7
OK
RUN
1000 BL 3 2000    3000    4000    5000    6000
-7000
```

The spaces that appear between the numbers are due to the fact that the system adds one space after each number and eliminates the implied "plus" sign before each positive value. If semi-colons and commas are mixed in the same PRINT statement, this provides a simple method of labelling results and of obtaining

appropriate spacing in a line.

For example:

```
20 PRINT "Base and Height"; B,H
20 PRINT "Area=";B*H,"Base=";B,"Height=";H
30 GOTO 10
OK
RUN
Base and Height? 1,2,3
Area= 3,6       Base=1,2       Height=3
Base and Height?^C
Break in 10
OK
```

If the special built-in function TAB is used, this determines the exact position in the line to which the cursor must be set for displaying the data.

For example:

```
PRINT a; TAB(6); 2
 1         2
OK
```

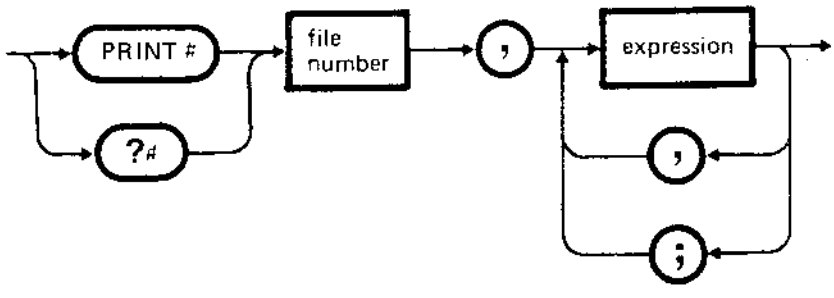
Note

The same statement may contain several "cursor" clauses.

PRINT

Writes data to a sequential file in the same standard format as used by the PRINT statement.

PROGRAM/IMMEDIATE Statement



where:

file number — is a numeric expression whose rounded value specifies the number of the file.

expression — may be a numeric, relational, logical or string expression whose value is to be written to the file.

Characteristics — When a PRINT # statement is executed, the data are written sequentially to the specified file.

If the file is opened in OUTPUT ("O"), the pointer is set to the beginning of the file; therefore the first PRINT # records data starting from the beginning of the file. At each PRINT # statement, the pointer moves forward, so the values are stored in sequence.

If the file is opened in Append ("A"), the file pointer is set to the end of the file; therefore the first PRINT # statement executed writes the data after the last data item in the file. At each PRINT # statement, the pointer moves forward so the data are stored in sequence.

If the PRINT # list is to be set up correctly for access using one or more INPUT # statements, it should be remembered that the PRINT # statement creates a string on disk that is similar to that created by PRINT on the screen.
PRINT # records a string of data coded in ASCII. The punctuation in the PRINT # list is very important. Commas and semi-colons have the same effect as in PRINT statements.

If numeric values are to be stored (resulting from the evaluation of a numeric, relational or logical expression), both commas and semi-colons may be used to separate the expressions.

Where string values are to be stored, explicit delimiters must be inserted if these values are to be read as distinct strings (using the INPUT # statement).

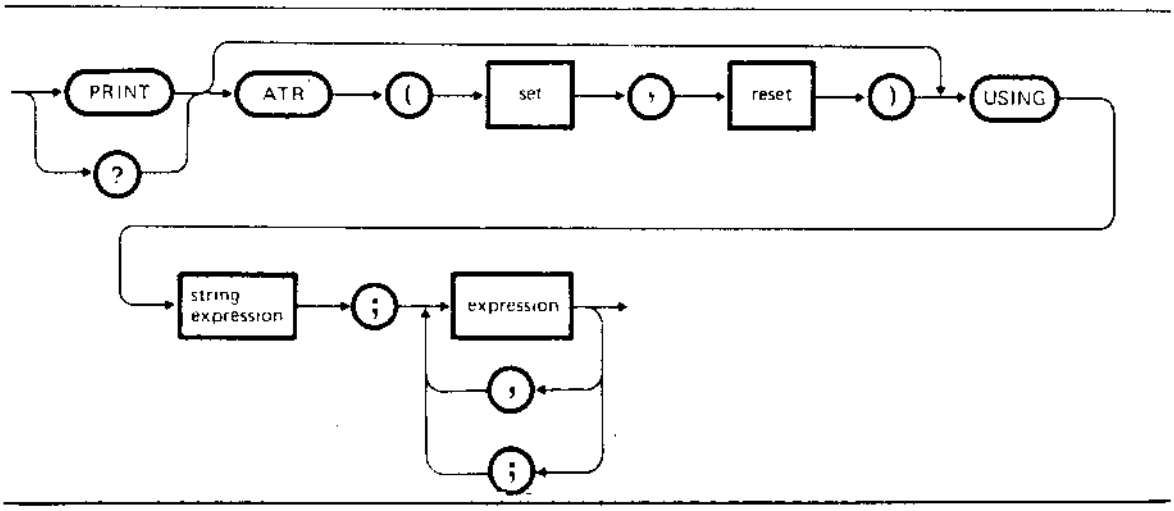
If string values not containing commas, semi-colons, significant leading or trailing blanks, carriage returns or line feeds are to be stored, a comma must be used as a string constant (",") to separate string expressions in the PRINT # statement. In this way, data items will be separated on the disk by commas and will be read back as distinct strings by INPUT # statement.

When string values are to be stored that contain commas, semi-colons, significant leading or trailing blanks, carriage returns and line feeds, the string expressions must be separated by quotation marks ("), expressed as the CHR$(34) function.

PRINT USING

Displays a list of data in a user-defined format.

PROGRAM/IMMEDIATE Statement



where:

set, reset	have the same functions as in the PRINT statement.
string expression	is a string of formatting characters or a string variable consisting of a string of formatting characters.
expression	is a numeric, relational, logical or string expression to be displayed.

Characteristics — Display of strings.

One of the following three formatting characters may be used:

"!"

specifies that only the first character in the given string is to be printed or displayed.

For example:

```
10 A$="WATCH"
20 B$="OUT"
30 PRINT USING "!";A$;B$
OK
RUN
WO
OK
```

"\n spazi\"

specifies that the first 2 + n characters of the string are to be displayed.
If the backslashes are entered with no spaces, two characters will be displayed.
If they enclose one space, three characters will be displayed and so on. If the string is longer than the field, the extra characters are ignored. If the field is longer than the string, the string will be left-justified in the field and right-filled with spaces.

For example:

```
10 A$="LOCK"
20 B$="OUT"
30 PRINT USING "\    \"; A$;B$
40 PRINT USING "\   \";A$;B$
OK
RUN
LOCKOUT
LOCK OUT
```

"&"

specifies a variable length string field. When the field is specified with "&", it is displayed exactly as input.

For example:

```
10 A$="LOOK":B$="OUT"
20 PRINT USING "!";A$
30 PRINT USING "&";B$
OK
RUN
LOUT
OK
```

Display of numbers.

The following formatting characters may be used:

#

a number sign is used to represent each digit position. Digit positions are always filled by digits or by space. If the number to by displayed has fewer digits than the positions specified, the number will be right-justified and preceded by spaces.

```
PRINT USING "####";99
  99
OK
```

.

A decimal point may be inserted in any position in the field. If the format string specifies that a digit is to precede the decimal point, the digit will be displayed only if other than 0. Numbers are rounded where necessary.

For example:

```
PRINT USING "# #.##";.78
   .78
OK

PRINT USING "##.##";987.654
987.65
OK

PRINT USING "##.## ";10.,5.3,66.789,.234
10.00   5.30  66.79    .23
OK
```

In the last example, one space have been inserted at the end of the format string to separate the values displayed on the line.

+

A plus sign at the beginning or end of a format string will cause the sign of the number (i.e. plus or minus) to be displayed before or after the number.

For example:

```
PRINT USING "+##.## ";-68.95,2.4,55.6,-.9
-68.95   +2.40  +55.60    -.90
```

If only the minus sign is to be displayed (and not the plus sign) before the number, an extra number sign ( ) should be inserted at the start of the format string.

For example:

```
PRINT USING "###.##";-68.95,68.95
```

```
-68.95  68.95
```

-

A minus sign at the end of the format string will cause negative numbers to be displayed with trailing minus sign.

For example:

```
PRINT USING "##.##- ";-68.95,22.449,-70
68.95-  22.45    70-
```

**

A double asterisk at the beginning of the format string will cause the initial spaces in the numeric field to be filled with asterisks. The ** also specifies positions for two more digits.

For example:

```
PRINT USING "**#.# "; 12.39,-09,765.1
*12.4  **-.9  765.1
OK
```

$$

A double dollar sign ($$) causes a dollar sign to be dispalyed immediately to the left of the formatted number. The $$ specifies two more digit positions one of which is a dollar sign. The $$ character cannot be used with the exponential format (^^^^). Negative numbers cannot be used unless the format string ends with a minus or plus sign. In the former case numbers appear with the negative sign on the right, in the latter case both positive and negative numbers appear with the appropriate sign on the right.

For example:

```
PRINT USING "$$###.##-";-456.78
 $456.78-
OK
```

**$

**S at the beginning of a format string combines the effect of the two symbols described above. Leading spaces will be filled with asterisks and a $ sign will be inserted immediately in front of the number.

**$ specifies three more digit positions.

For example:

```
PRINT USING "**$##.##"; 2.34
***$2.34
OK
```

A comma to the left of the deciaml point in a formatting string causes a comma to be displayed to the left of each third digit in the integer part of the figure. A comma specifies another digit position. A comma has no effect if used with the exponential format (^^^^).

For example:

```
PRINT USING "####,.##";1234.5
1,234.50
OK
```

with a comma at the end of the format.

For example:

```
PRINT USING "####.##,";1234.5
1234.50,
OK
```

Four carats (or up arrows) may be placed after the digit position characters to specify exponential format. These symbols allow space to be generated to display E+xx. Any deci l point position may be specified.

Significant digits are left-justified and the exponent is adjusted accordingly.
Unless a leading + or trailing + or - is specified, one digit position will be used to the left of the decimal point to display a space or a minus sign

For example:

```
PRINT USING "##.##^^^^";234.56
 2.35E+02
OK

PRINT USING ".####^^^^";888888
 .8889E+06
OK

PRINT USING "+.####^^^^";123
+.1230E+03
OK
```

An underscore in the format string causes the next character to be displayed as a literal character (i.e. as it appears in the format string).

For example:

```
PRINT USING "_!##.##_!";12.34
!12.34!
OK
```

The literal character itself may be underscored placing "__" in the format string.

%

If the number to be displayed is longer than the specified numeric field, a percent sign is displayed in front of the number. If rounding causes the number to exceed the field, a percent sign will be displayed in front of the rounded number.

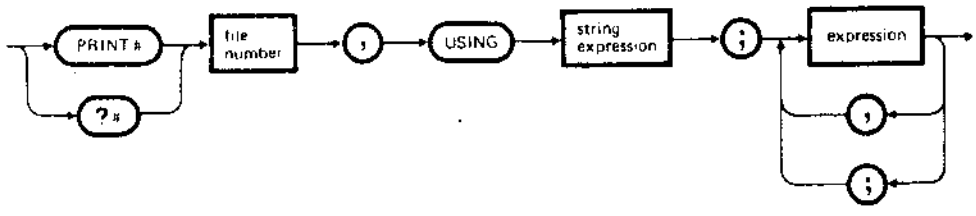
For example:

```
PRINT USING "  ##.##";111.22
%111.22
OK

PRINT USING ".##";.999
%1.00
OK
```

PRINT # USING

Writes data to a sequentail file in a user-defined format in the same way as the PRINT USING statement displays data on the screen.

PROGRAM/IMMEDIATE Statement



where:

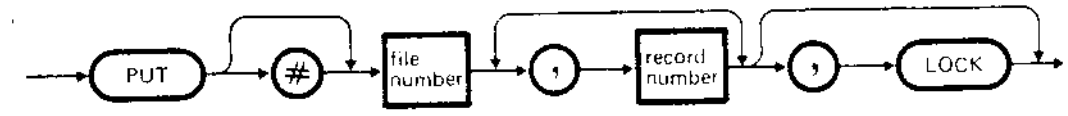
file number	is a numeric expression whose rounded value specifies the file number.
string expression	see PRINT USING.
expression	can be a numeric, relational, logical or string expression whose value is written to the file.

PUT

Format 1

Writes data from a random file buffer to a random file.

PROGRAM/IMMEDIATE Statement



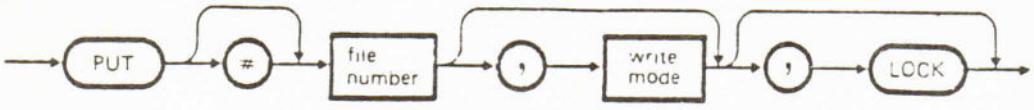
where:

file number	is a numeric expression which specifies the number of the file
record number	is a numeric expression which specifies the record number in the file. The lowest record number is 1, the highest 32767. If record number is not specified, the current record is written. The current record is the record whose number of one higher than the number of the last record accessed. The first time a random file is accessed, the current record is set to 1.
LOCK	Optional keyword that allows the record just written to be LOCKed. If LOCK is not specified, any previous LOCK associated with that record is removed.

Format 2

Writes data from a keyed file buffer to a keyed file.

PROGRAM/IMMEDIATE Statement



where:

file number — is a numeric expression specifying the number of the file.

write mode — This parameter may have the following values:

0= rewrite (checks and re-writes an existing record, to carry out a check)

1= newwrite (writes a new record)

2= alwayswrite (always writes, both a new or an old record)

The default value is 2.

LOCK — Optional keyword that allows the record just written to be LOCKed. If LOCK is not specified, any previous LOCK on that record will be removed.

Example 1

```
10 OPEN "r",1,"RAND",48
20 FIELD 1,20 AS R1$,20 AS R2$,8 AS R3$
30 FOR L=1 TO 4
40 INPUT "name";N$
50 INPUT "address";M$
60 INPUT "phone";P#
70 LSET R1$=N$
80 LSET R2$=M$
90 LSET R3$=MKS$(P#)
100 PUT 1,L
110 NEXT L
120 CLOSE 1
130 END
OK
RUN
name? super man
address? USA
phone? 11234621
```

```
name? robin hood
address? England
phone? 23462101
    .
    .
    .
OK
```

Statement 10 opens the direct file RAND with a record length of 48 bytes. The file number is 1. Statement 20 divides the buffer into fields.

Statement 100 writes a record in a file RAND. The record number can be defined using the control variable of the FOR/NEXT interactive cycle.

Example 2

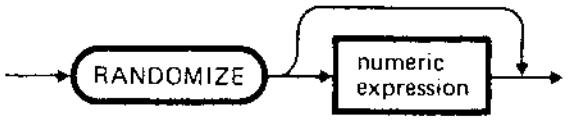
```
   ...
   ...
70 OPEN "k",#1,F$ 20
80 FIELD #1,5 AS A$ KEY(1),5 AS B$ KEY(2),10 AS C$
90 INPUT "1=write 3=exit"; K$
100 IF K$="3" THEN 300
    ...
    ...
130 INPUT "0=rw 1=nw 2=aw "; W
140 INPUT "record write "; AA$,BB$,CC$
150 LSET A$=AA$:LSET B$=BB$:LSET C$=CC$
160 PUT #1,W
170 GOTO 90
    ...
    ...
390 CLOSE
400 END
```

Statement 70 opens the keyed file whose name is the content of the string variable F$. The record length is 20 byte. File number is 1. Statement 80 divides the buffer into fields. Statement 160 writes a record in the file.

RANDOMIZE

Reinitialises the random number generator.

PROGRAM/IMMEDIATE Statement



where:

numeric expression

must be in the range from -32768 to 32767. If its value is not an integer, it is rounded to the nearest integer. This number is used to define the starting point of a new sequence of random numbers. If it is omitted, program execution is interrupted and a new value is requested with the following message:

Random Number Seed (-32768 to 32767)?

before executing the RANDOMIZE statement.

Characteristics

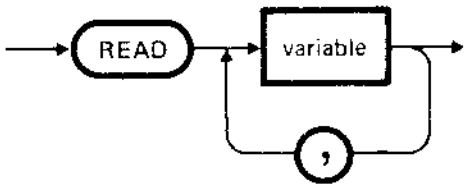
If the random number generator is not reinitialised, the RND function returns the same sequence of random numbers each time the program is run. To modify the sequence of random numbers a RANDOMIZE statement must be inserted at the beginning of the program itself and the argument must be modified each time the program is run.

Example

```
10 RANDOMIZE
20 FOR I=1 TO 5
30 PRINT RND;
40 NEXT I
RUN
 :
```

Reads the data of the internal file (created by one or more DATA statements) and assigns them to the specified variables.

PROGRAM Statement



Characteristics

READ statements can contain both numeric and string variables.

Example 1

```
10 READ A,B,C,D,E,F,G,H,I,J
20 DATA 1,2,3,4,5,6,7,8,9,10
30 PRINT A;B;C;D;E;F;G;H;I;J
OK
RUN

 1  2  3  4  5  6  7  8  9  10

OK
```

The values 1 to 10 are assigned to ten variables.

Example 2

```
10 DATA 1,2,3,4
20 READ A,B,C,D,E,F,G,H,I,J
30 DATA 5,6,7
40 DATA 8,9,10
50 PRINT A;B;C;D;E;F;G;H;I;J
OK
RUN
```

```
 1  2  3  4  5  6  7  8  9  10

OK
```

Statements 10, 20, 30 and 40 have the same effect as statements 10 and 20 in the above example.

Example 3

```
10 READ A,B,C
20 DATA 1,2,3,4,5,6,7,8,9,10
30 PRINT A;B;C
40 READ D,E,F,G
50 PRINT D;E;F;G
RUN

 1  2  3
 4  5  6  7

OK
```

Statement 10 assigns the values 1, 2 and 3 to A, B and C.
Statement 40 assigns the values 4,5,6, and 7 to D,E,F and G respectively.

Example 4

```
10 READ X1$,Y1$,Z1
20 DATA "DENVER,", COLORADO, 80211
30 PRINT X1$;Y1$;Z1
OK
RUN
DENVER,COLORADO 80211
OK
```

Statement 10 causes the value DENVER to be assigned to X1$ (including the final comma and the value) COLORADO to Y1$ and 80211 to Z1.

Example 5

```
10 READ A,B,C
20 DATA 15,25,35,5,6,12
30 PRINT A;B;C
40 RESTORE
50 READ X,Y,Z
60 PRINT X;Y;Z
OK
RUN

 15  25  35
 15  25  35

OK
```

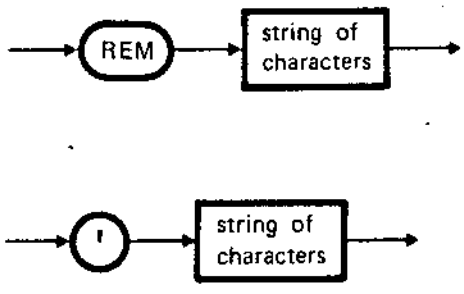
Statement 10 causes the value 15 to be assigned to variable A, 25 to variable B and 35 to variable C.

The RESTORE statement at line 40 will cause values to be assigned starting from the beginning of the file again. Then, statement 60 causes the values assigned to A,B,C (15,25,35) to be assigned to M,Y,Z.

If RESTORE were not used, 5 would be assigned to X, 6 to Y and 12 to Z.

The REM statement can be used to document a program. Any string of characters can be written after the REM keyword.
A comment field may also be inserted i.e. a string of characters preceded by an apostrophe (') and closed with a carriage return/line feed.

PROGRAM Statements



Characteristics

A REM statement cannot be used on a line with several statements.

A comment field may:

- fill an entire line (in this case, the apostrophe has the same function as REM)
- end a statement.

Both REM and command fields may be inserted anywhere in a program. They are not executable statements but they appear in the listing.

Example 1

```
10 REM RETTANGOL01
```

A REM statement is used to give a title to a program. It is good programming practice to give a title to a program.

Example 2

```
100 REM SUBROUTINE1
```

A REM statement marks the start of a subroutine. It is a good programming practice to give a title to a subroutine.

Example 3

```
10 RETTANGOL01
```

A comment field is used to give a title to a program.
In this case, the apostrophe has the same function as the REM keyword as the comment occupies an entire line.

Example 4

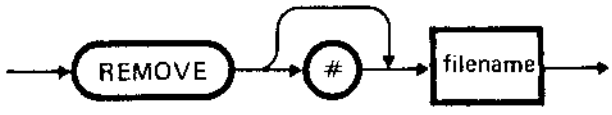
```
150 LET A=(A1+A2)/2 'Average
```

A comment field is used to end a statement.

REMOVE

Deletes a record from a keyed file.

IMMEDIATE Command.



where:

file number — is the file number used when the file is opened.

Characteristics — The record identified by the primary key (with the FIELD statement) is deleted from the file.

Example 1

```
OPEN "K",#1,"DATA"
FIELD #1,20 AS N$ KEY(1),4 AS A$,30 AS D$ KEY(2)
LSET N$ ="SMITH"
REMOVE #1
```

The record identified by the primary key "SMITH" is deleted.

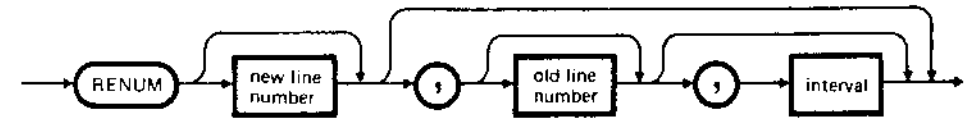
Example 2

```
   ...
   ...
70 OPEN "k",#1,F$,20
80 FIELD #1,5 AS A$ KEY(1),5 AS B$ KEY(2),10 AS C$
90 INPUT 2=delete 3=exit ";K$
100 IF K$="3" THEN 390
    ...
    ...
350 INPUT "key input 11=5 ";AA$
360 LSET A$=AA$
```

```
370 REMOVE #1
380 GOTO 90
390 CLOSE
400 END
```

Changes the line numbers of the current program.

IMMEDIATE Command



where:

new line number	is the new first line number. The defualt value is 10.
old line number	the old first line number to be modified. If this is not specified, the first line number of the program is assumed by default.
interval	the new interval between line numbers. The default value is 10.

Characteristics — When a program is renumbered with RENUM, all the line numbers referred to after the GOTO, GOSUB, THEN, ELSE, ON...GOTO, ON...GOSUB are automatically updated.

Example 1

```
RENUM
```

The entire program is renumbered.
The first new line number is 10 with a line interval of 10. (Default value)

Example 2 RENUM 100

The entire program is renumbered.
The first new line number of 100 with an interval of 10. (Default value)

Example 3 RENUM 150,,20

The entire program is renumbered.

The first new line number is 150 with a line interval of 20.

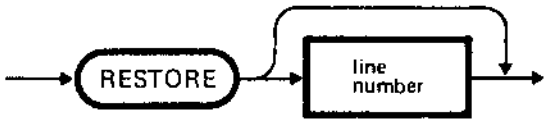
Example 4 RENUM 1000,900,20

The lines starting at line 900 are renumbered, starting with new line number 1000 with a line interval of 20.

RESTORE

Moves the pointer to the start of an internal data file or to the specified line number.

PROGRAM Statement



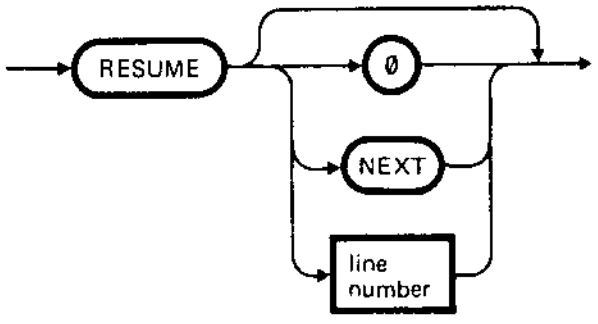
Examples

See READ Statement examples.

RESUME

Resumes execution of the program after execution of an error handling routine.

PROGRAM/IMMEDIATE Statement



where:

0	execution will restart at the statement that caused the error. RESUME 0 and RESUME are equivalent.
NEXT	execution will restart from the first statement after the one that caused the error.
line number	execution will restart from the specified line number.

Example 1

```
10 REM RECTANGLE3
20 ON ERROR GOTO 70
30 INPUT "Base and Height";B,H
40 IF (B<0) OR (H<0) THEN ERROR 200
50 PRINT "Area=";B*H;"B=";B;" H=";H
60 GOTO 30
70 IF (ERR=200) AND (ERL=40) THEN RESUME
```

```
80 ON ERROR GOTO 0
90 END
OK
RUN
Base and Height? -2,5
^C
OK
```

If a negative value is entered for B or H, the error handling routine is activated and the system restarts execution from the statement that caused the error thus causing a deadlock.

/CTRL/ /C/ must be entered to interrupt execution.

Example 2

```
70 IF (ERR=200) AND (ERL=40) THEN RESUME NEXT
RUN
Base and Height? -2,5
Area=-10 B=-2 H= 5
Base and Height?   ^C
Break in 30
OK
```

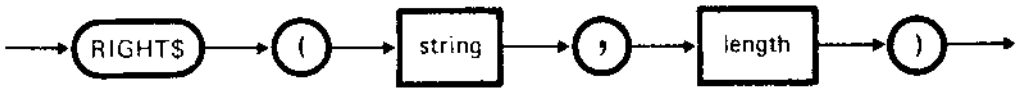
The error is ignored correcting line 70 as follows:

```
70 IF (ERR=200) AND (ERL=40) THEN RESUME 30
RUN
Base and Height? -2,5
Base and Height? 2,5
Area= 10 B= 2 H= 5
Base and Height?^C
Break in 30
OK
```

If line 70 is corrected in this way, the error handling routine will restart execution from line 30.

RIGHT$

This function returns a substring of the specified length extracting it starting from the rightmost character of a specified string.



where:

string — is a string expression whose value is the original string from which the substring is to be extracted.

length — is a numeric expression rounded to the nearest integer whose value (from 0 to 255) represents the length of the substring to be extracted.

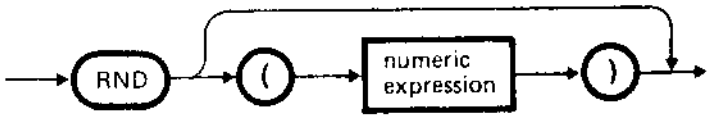
Characteristics — If the value of "length" is 0, the null string is returned.
If length ≧ LEN(string), the entire original string is returned.

Note — See the MID$ and LEFT$ functions.

Example

```
10 A$="DISK BASIC"
20 PRINT RIGHT$(A$,5)
RUN
BASIC
OK
```

This function returns a random number between 0 and 1.



where:

numeric expression

<0 restarts the same random number sequence.
=0 repeats the last number generated.
>0 (or omitted) the next random number in the seqence is generated

Characteristics

The same sequence of random numbers is generated each time the program is run unless the random number generator is reinitialised (with the RANDOMIZE statement). Although it is referred to as random, the number is actually taken from a fixed cycle of numbers (approx. 1 million).

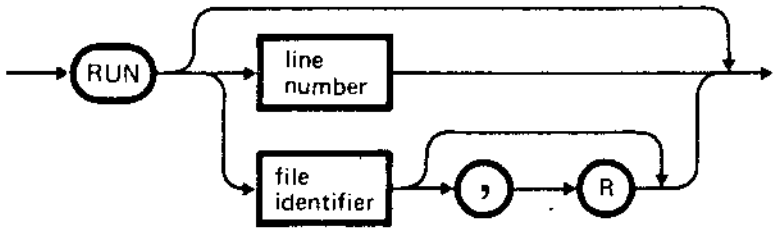
Example

```
10 FOR I=1 TO 5
20 PRINT INT(RND*100);
30 NEXT
RUN
 8   25   77   68   7
OK
```

RUN

Runs the current program. If the RUN command specifies the name of a disk program file this is loaded into the memory and executed.

PROGRAM/IMMEDIATE Command



where:

line number	specifies the entry point of the program i.e. the line number at which execution starts. If line number is not specified, the program is executed starting from the first statement.
file identifier	may be either a string constant or a string variable. It specifies the program file to be loaded from disk into the memory and executed. The value of the file identifier must match the one used when the program was written to disk.
R	this option specifies that all the open data files of the previously executed program must remain open. If R is not specified, the data files are automatically closed before execution of the new program begins.

Characteristics

The RUN command also sets all the variables to zero (i.e. it sets the numeric variables to 0 and assigns the null string to the string variables). After execution of the program with the RUN command, the system returns to the Command State.

Example 1

```
RUN
```

The current program is executed.

Example 2

```
RUN 150
```

The current program is executed starting from statement 150.

Example 3

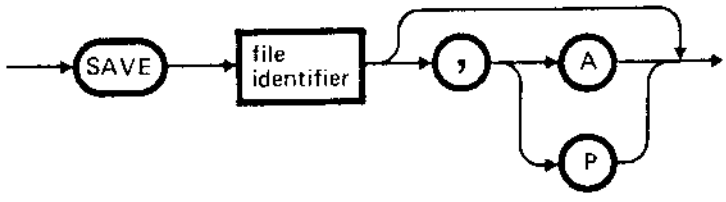
```
RUN "Newfile"
```

The Newfile program is loaded from disk into memory and executed.

SAVE

Stores the current program on disk.

PROGRAM/IMMEDIATE Command



where:

file identifier	may be either a string constant or a string variable, which specifies the name of the program to be saved on disk.
A	specifies that the program must be saved in ASCII format. If A is not specified, the file is recorded in compact binary format.
P	specifies that the program must be saved on disk, protected against attempts to list, edit or save it again.

Note — ASCII format takes up more disk space than the "compact" binary format but some commands (e.g; the MERGE command) can be performed only on files in ASCII format.

Example 1

```
SAVE "GEOMETRY",A
```

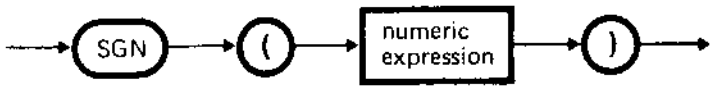
The GEOMETRY program is recorded in ASCII format (i.e. as a sequence of ASCII characters).

Example 2

```
SAVE "GEODESY",P
```

GEODESY is recorded in compact binary format and protected.

This function returns 1 if the argument is positive, 0 if the argument is zero and -1 if the argument is negative.



Example

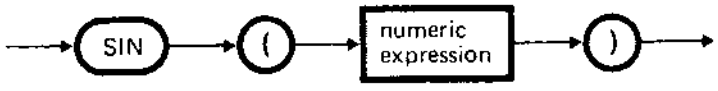
```
ON SGN(X)+2 GOTO 100,200,300
```

branches to:

100 if X<0
200 if X=0
300 if X>0

SIN(SINUS)

This function returns the sine of the argument.



Characteristics

The argument passed to the function is assumed to be the value of an angle measured in radians.
SIN is evaluated in double precision.

Note

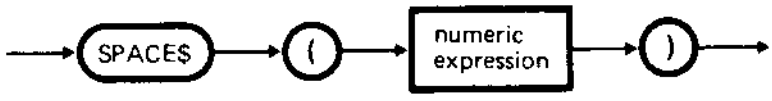
The numeric expression may be of any type. The result provided is in double precision. It is advisable to use a double precision numeric expression as input.

Example

```
PRINT SIN(1.5)
  .997494986604054
OK
```

SPACE$

This function returns a string of spaces. The length of the string is specified by the user.



where:

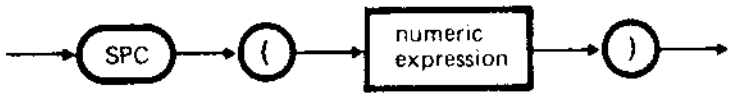
numeric expression — is rounded to the nearest integer and must be in the range from 0 to 255.
It indicates the number of spaces that the string must contain.

Note — See the SPC function.

Example

```
10 FOR I=1 TO 5
20 X$ = SPACE$(I)
30 PRINT X$;I
40 NEXT I
RUN
  1
   2
    3
     4
      5
OK
```

This function inserts spaces. It can be used only with the PRINT and LPRINT statements.



where:

numeric expression — is rounded to the nearest integer. It specifies the number of spaces to be inserted in the image displayed either between two data items or at the start or end of the image.

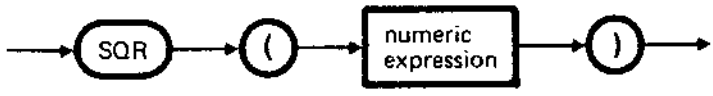
Note — See the SPACE$ function.

Example

```
PRINT "OVER" SPC(15) "THERE"
OVER                 THERE
OK
```

SQR(SQUARE ROOT)

This function returns the square root of the argument.



Characteristics

The argument must be greater than or equal to 0. The square root is retured in double precision.

Note

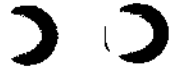
The numeric expression can be of any type. The result provided is in double precision. It is advisable to use a double precision numeric expression as input.

Example

```
10 FOR X=10 TO 25 STEP 5
20 PRINT X, SQR(X)
30 NEXT
RUN
 10          3.16227766016838
 15          3.87298334620742
 20          4.47213595499958
 25          5
OK
```

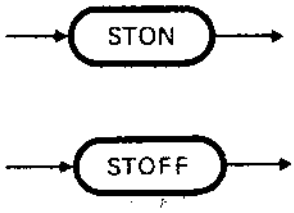






Enables/disables step by step execution of a program.

PROGRAM/IMMEDIATE Command



Characteristics

Allows step by step execution of a program. After each single statement has been executed, the program stops and the system returns to the Command State. To continue execution of the next program statement, line feed key or the CONT command must be entered. Loading of a program (LOAD, RUN, CHAIN) automatically resets TRACING.

Example

```
10  FOR I=1 TO 5
20  PRINT I
30  NEXT
40  END
    STON
    RUN
   *10*
    OK
   *20*
    1
    OK
   *30*
```

```
 OK
*10*
 OK
*20*
 2
 OK
 STOFF
 CONT
  3
  4
  5
 OK
```

STOP

Interrupts program execution temporarily and returns the system to the Command State.

PROGRAM/IMMEDIATE Statement

Characteristics

Like the END statement, STOP can be used anywhere in a program. When a STOP statement is encountered, the following message is displayed.

```
Break in nnnnn
```

When a STOP statement has been executed, the values of the program variables can be displayed using the PRINT (or PRINT USING) statement, execution can then be resumed with the CONT statement.
Unlike the END statement, STOP does not close files.

Example

```
10 INPUT A,B,C
20 X=A*B
30 STOP
40 X=X/C
50 PRINT X
60 END
OK
RUN
? 4,3,6
Break in 30
OK
PRINT X
 12
OK
CONT
 2
```

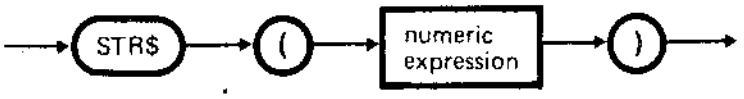
OK

Statement 30 allows the first value of X to be checked and observed before the second is calculated and displayed.
Although in such a simple case, the STOP statement does not appear very useful, it can be very effective when used in larger programs. When a STOP statement is entered at the end of a branch, the program will stop only if the branch is used. It also allows certain variables to be modified before program execution is resumed with the CONT. statement.

When the program has been sufficiently tested, all the STOP statements inserted for debugging can be deleted and the program can be renumbered.

STR$(STRING$)

This function converts a numeric expression to a string.



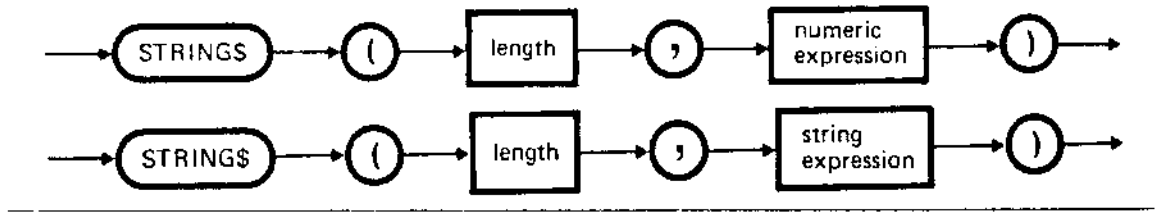
Note

See the VAL function (it carries out the reverse operation).

Example

```
10 A$=STR$(70)
20 PRINT A$
 70
OK
```

This function returns a string of specified length whose characters are either the same as the character whose ASCII code has been specified or as the first character of a specified string.



where:

length	is a numeric expression rounded to the nearest integer. It specifies the length (from 0 to 255) of the resulting string.
numeric expression	is rounded to the nearest integer. It specifies the ASCII decimal code (from 0 to 255) whose corresponding character is to be used to form the returned string.
string expression	is examined. Its first character is used to form the string requested.

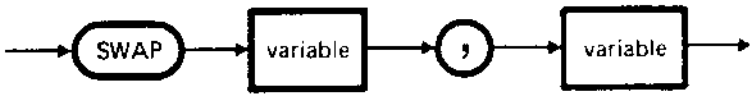
Example

```
10 X$=STRING$(10,45)
20 PRINT X$"MONTHLY REPORT"X$
RUN
---------MONTHLY REPORT----------
OK
```

SWAP

Swaps the value of two variables.

PROGRAM/IMMEDIATE Statement



Characteristics

Any type of variable can be swapped (integer, in single precision, in double precision, string) but the two variables must be of the same type and already initialized.

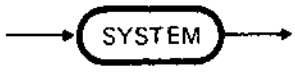
Example

```
10 A$ = " ONE "
20 B$ = " ALL "
30 C$ = " FOR "
40 PRINT A$ C$ B$
50 SWAP A$, B$
60 PRINT A$ C$ B$
RUN
OK
 ONE FOR ALL
 ALL FOR ONE
OK
```

Statement 50 swaps the values of A$ and B$.
Statement 40 displays ONE FOR ALL while statement 60 displays ALL FOR ONE.

Closes the BASIC session and returns to the SHELL environment.

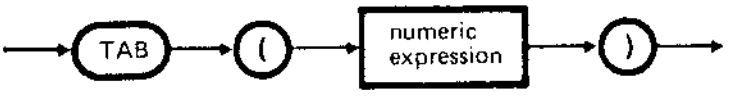
PROGRAM/IMMEDIATE Command.



Characteristics

The SYSTEM command is used to return to the SHELL environment. It can be used in immediate mode or inside a BASIC program.
All files are closed before returning to SHELL.

When used in a PRINT or LPRINT statement, the TAB function positions the cursor or the printhead to the specified position.



where:

numeric expression

is rounded to the nearest integer.
The minimum allowed value is 1, while the maximum value is the length of the line -1. If the value of numeric expression exceeds the value already defined for the line (see the WIDTH command), the arithmetic module is returned by dividing the latter by the value of the numeric expression.
This value specifies the position of the cursor (or printhead) on a line.

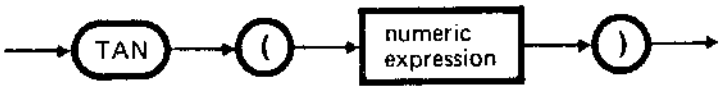
Characteristics

If the position of the cursor or printhead is beyond the position indicated by the value of the argument, TAB positions the cursor or printhead to the correct position on the next line.

Example

```
10 PRINT "NAME" TAB(25) "AMOUNT":PRINT
20 READ A$,B$
30 PRINT A$ TAB(25) B$
40 DATA "G.T.JONES","$25.00"
RUN
NAME                      AMOUNT
G.T.JONES                  $25.00
OK
```

This function returns the tangent of the argument.



Characteristics	The argument of the function is the measurement of an angle in radians. The tangent is returned in double precision.
Note	The numeric expression can be of any type. The result provided is in double precision. It is advisable to use a double precision numeric expression as input.
Example	10 Y=Q*TAN(X)/2

TIME$

This function allows the time to be set and read.

Characteristics

The time is set or read in hours, minutes, seconds. Any delimiter can be used when setting the time (any printable ASCII character except for digits). The delimiter in output is a slash.

Example

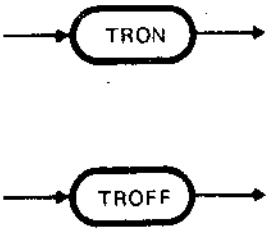
```
700 TIME$ = "07:40:15"
      .
      .
      .
      .
      .
      .
750 PRINT TIME$
```

TRON/TROFF

TRON causes the line numbers of each statement executed to be listed.

TROFF stops the listing initiated by TRON.

PROGRAM/IMMEDIATE Command



Characteristics

The line numbers of the statements executed are dispalyed between asterisks.
The list of numbers can be interrupted not only with the TROFF command but also using the NEW command (in this case, the program is deleted).
Loading of program (LOAD, RUN, CHAIN) automatically resets TRACING.

Example

```
10 K=10
20 FOR J=1 TO 2
30 L=K+10
40 PRINT J;K;L
50 K=K+10
60 NEXT
70 END
OK
TRON
OK
RUN
*10**20**30**40* 1   10   20
*50**60**30**40* 2   20   30
```

```
*50**60**70*
OK
TROFF
OK
RUN
 1   10   20
 2   20   30
OK
```

TRON displays the line number of each statement executed. The numbers are displayed between asterisks. Numbers that are not between asterisks are the result of statement execution.

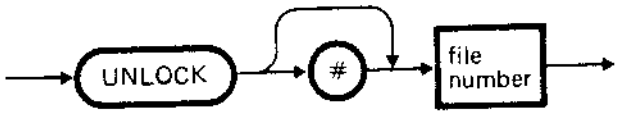
```
40 PRINT J;K;L
```

Display of the line numbers of the executed statements is interrupted with the TROFF command or with the NEW command (in this case, the program is deleted. It must therefore be re-entered if it is to be re-executed).

UNLOCK

Frees all the LOCKs set previously with the GET LOCK statement.

PROGRAM/IMMEDIATE Statement



where:

file number — is a numeric expression specifying the number of the file.

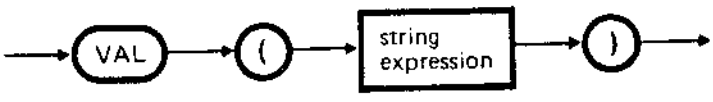
Example

```
10 OPEN "K", #1, "PIPPO"
20 FIELD #1, 10 AS A$ KEY(1), 5 AS B$
   ...
   ...
50 GET #1, KEY(1), LOCK
   ...
   ...
100 PUT  #1,L, LOCK
    ...
    ...
170 UNLOCK #1
```

Statement 10 opens a keyed data file with the name PIPPO and assigns buffer 1 to it.
Statement 20 assigns the space for the variables specified with buffer of the keyed file which was opened with statement 10. It also specifies fields to be used as keys.
With statements 50 and 100, the reading and storing

of a file can be carried out on the file PIPPO.
Use of the LOCK clause in these two statements prevents any other user from updating and reading the file.
Statement 170 cancels are the LOCK clauses previously entered in file 1.

This function converts a string expression into a numeric expression.



where:

string expression — is the string expression to be changed into a number. Any leading or trailing spaces, tabulation or line feed characters are eliminated. If the expression contains even one alphabetic character, the function returns the numeric value 0.

Note — See the STR$ function (carries out the reverse operation).

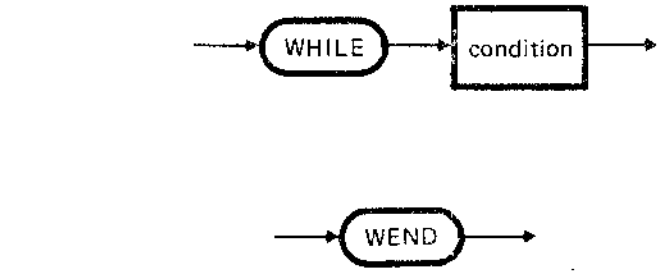
Example

```
10 READ NAME$, CITY$, STATE$, ZIP$
20 IF VAL(ZIP$)<90000 OR VAL(ZIP$)>96699 THEN
   PRINT NAME$ TAB(25) "OUT OF STATE"
30 IF VAL(ZIP$)>=90801 AND VAL(ZIP$)<=90815 THEN
   PRINT NAME$ TAB(25) "LONG BEACH"
```

WHILE/WEND

Executes a number of statements in a loop until a given condition remains true.

PROGRAM/IMMEDIATE Statement



where:

condition

may be:

- a numeric expression
- a relational expression
- a logical expression.

The BASIC program establishes whether the condition is true or false by testing the result of the expression for zero and non zero respectively. A non zero result is true and a zero result is false. This means that the value of a variable can be tested for zero or non zero merely specifying the name of the variable as a condition.

Characteristics

WHILE/WEND loops can be nested to any level. Each WEND will match the most recent WHILE.
It is possible to exit from a WHILE/WEND loop when the condition after WHILE is false or using an IF...THEN or GOTO statement.

It is not possible to enter a WHILE/WEND loop without executing WHILE statement.

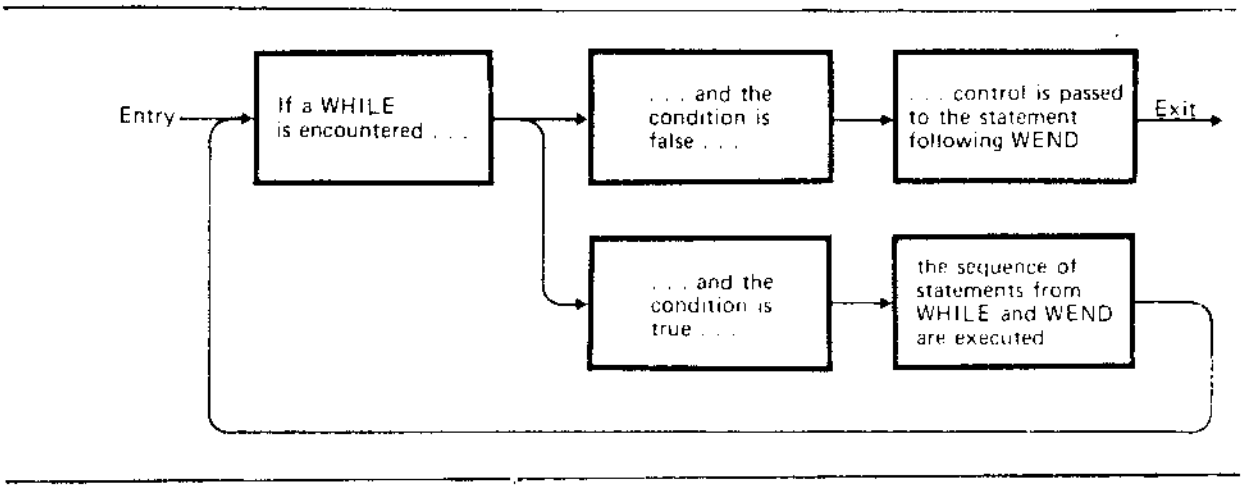


Fig. 1 - Function of the WHILE/WEND Statements

Example

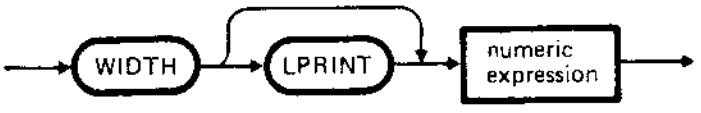
```
90 'BUBBLE SORT ARRAY A$
100 FLIP$=1 'FORCE ONE PASS THRU LOOP
110 WHILE FLIP$
115 FLIP$=0
120 FOR I=1 TO J-1
130 IF A$(I)>A$ (I+1) THEN
    SWAP A$(I),A$(I+1):FLIP$=1
140 NEXT I
150 WEND
OK
RUN
OK
```

The elements of array A$ are stored in ascending order (from subscript 1 to subscript J).

WIDTH

Sets the maximum width (in line characters) of the printer line.

PROGRAM/IMMEDIATE Command



Characterisitcs

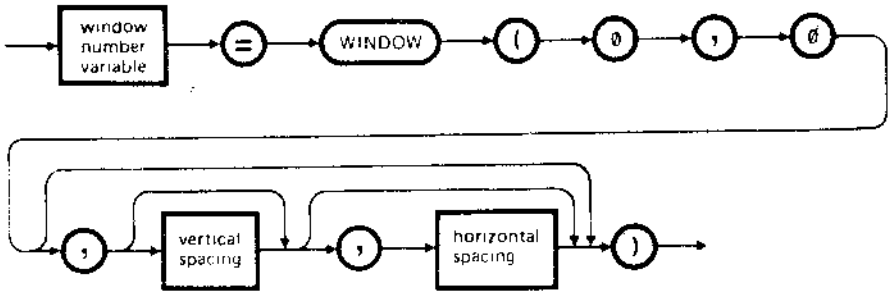
If the parameter LPRINT is included, the width of a print line is specified.

Numeric expression must be between 15 and 255. If it is equal to 255, the length of the line is 'infinite' i.e. BASIC never inserts a carriage return. However, the position of the cursor or the printhead, as given by the POS and LPOS functions returns to zero after position 255.

Similarly if WIDTH LPRINT is not specified, a print line width of 132 characters is assumed. The WIDTH command without the parameter LPRINT has no effect, neither does it cause an error message.

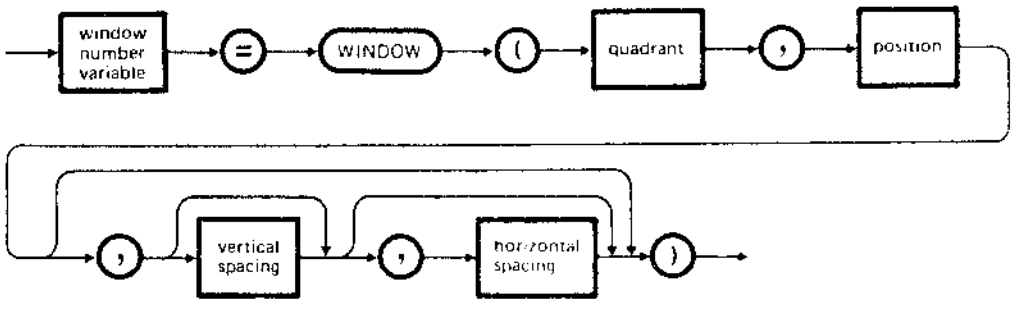
WINDOW

No new window is opened with this statement. It is accepted only for reasons of compatibility.



WINDOW (Def)

This function opens a new window by splitting the current window. The "current window" is the one being worked on.



where:

window number variable — is an integer variable to which BASIC assigns a value which identifies the window being opened. This value must be within the range 1 - 16. Values are assigned in ascending numeric sequence. The lowest number is always assigned to the new window. The window that initially covers the whole screen is always open and is identified with the number 1 even if it is later split into other windows.
A window that is subdivided into other windows is always known as the "parent"window.

quadrant — specifies in which part of the parent window a new window is to be opened.

There are four options:

0: top section of the parent window (TOP)
1: bottom section (BOTTOM)
2: left-hand section (LEFT)
3: right-hand section.(RIGHT)

position — defines the vertical or horizontal position where the parent window is to be split to open a new window.

If the value of quadrant is 0 (TOP) or 1 (BOTTOM), a horizontal split will be made.
In this case, 'position' is an integer number of scanlines calculated starting from the top of the current window (16 scanlines for each line of text). This number is a multiple of 16; otherwise, the number is automatically cut short to a nearest lower multiple of 16.

For example: position =32 specifies two lines of text starting from the top of the current window. Position =34 also specifies two lines of text.

If "quadrant" is equal to 2 (LEFT), or 3 (RIGHT), a vertical split is made. In this case, "position" is an integer number of pixels calculated starting from the left-hand edge of the current window (8 pixels for each character). This number must be a multiple of 8, otherwise it will be cut short to the nearest lower multiple of 8.

Position =-1 splits the parent window in half vertically or horizontally depending on the value of quadrant.

vertical spacing — this parameter is accepted only for reasons of compatibility.

horizontal spacing — this parameter is accepted only for reasons of compatibility.

Example

```
W2= WINDOW(1,205)
W3= WINDOW(0,40)
W4= WINDOW(2,20)
W5= WINDOW(3,35)
```

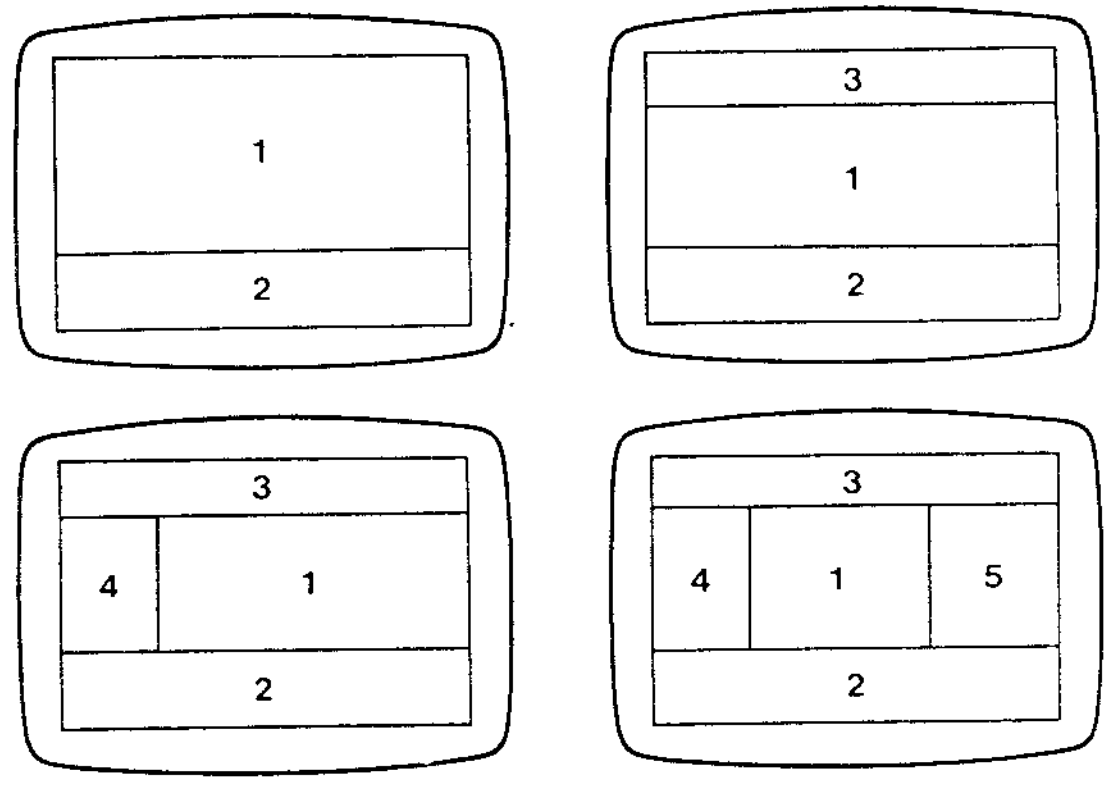
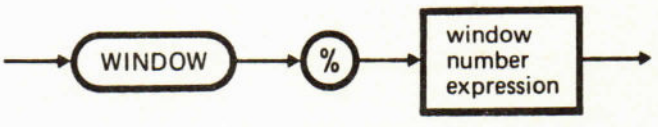


Fig. 2 - Order in Which Windows Are Opened

WINDOW (Sel)

Selects the window to be worked on. The selected window becomes the current window.

PROGRAM/IMMEDIATE Statement



where:

window number expression — is a numeric expression whose value rounded to the nearest integer identifies a window opened previously and selects it.
The window number expression parameter must be an integer between 1 and 16.

Example 1

```
WINDOW %A
```

The window identified by the integer value contained in variable A is selected. If this value is known, (e.g. 2) the WINDOW statement can be entered as follows:

```
WINDOW %2
```

Example 2

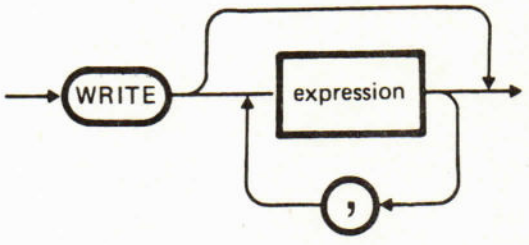
```
WINDOW %1
```

The window identified by the number 1 is selected. As previously mentioned, this is the parent window.

WRITE

Displays a data list. Each element of the list is separated from the next by a comma. Strings are delimited by quotation marks ("). After the last data item, BASIC executes a carriage return/line feed.

PROGRAM/IMMEDIATE Statement



where:

expression — can be a numeric, relational, logical or string expression. If expression is omitted, the WRITE statement inserts a line feed.

Example

```
10 A=80 : B=90
20 C$="THAT'S ALL"
30 WRITE A,B,C$
RUN
 80, 90, "THAT'S ALL"
OK
```

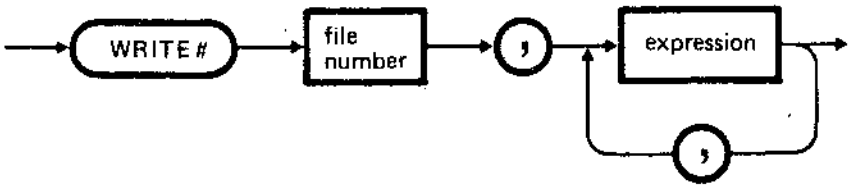
When the WRITE statement is executed, each data item is separated from the next by a comma and strings are delimited by quotation marks (").

The WRITE statement displays numeric values using the same format as the PRINT statement but these values will not be followed by spaces.

WRITE

Writes data to a sequential file using the same format as the WRITE statement. Each data item is separated from the next by a comma. Strings are delimited by quotation marks ("). After the last data item has been written, BASIC executes a carriage return and line feed.

PROGRAM/IMMEDIATE Statement



where:

file number — is a numeric expression whose rounded value specifies the file number.

expression — may be a numeric, relational, logical or string expression whose value is written to the file.

Note — Delimiters need not be inserted explicitly in the list of a WRITE # statement
If a string containing quotation marks is to be written, the PRINT # statement must be used instead of WRITE #.

Example

```
10 OPEN "O",1,"DATA2"
20 A$="CAMERA"
30 B$="93605-2"
40 WRITE#,A$,B$
50 CLOSE#1
60 OPEN "I",1,"DATA2"
70 INPUT#1,A$,B$,
80 WRITE A$,B$
90 CLOSE#1
100 END
OK
RUN
"CAMERA","93605-2"
OK
```

Statement 40 writes the following data to disk:

"CAMERA","93605-2"

Statement 70 assigns "CAMERA" to variable A$ and "93605-2" to B$. This can be tested with statement 80.

APPENDIX A. ERROR CODES

This appendix provides a list of error codes associated to the corresponding message.

ERROR CODE	MEANING
1	NEXT without FOR
2	Syntax error
3	RETURN without GOSUB
4	Out of DATA
5	Illegal function call
6	Overflow
7	Out of memory
8	Undefined line number
9	Subscript out of range
10	Duplicate definition
11	Division by zero
12	Subscript out of range
13	Type mismatch
14	Out of string space
15	String too long
16	String formula too complex
17	Can't continue
18	Undefined user function
19	No RESUME
20	RESUME without error
22	Missing operand
23	Line buffer overflow
26	FOR without NEXT
29	WHILE without WEND
30	WEND without WHILE

35	Window not open
36	Unable to create window
37	Invalid action-verd
38	Parameter out of range
39	To many dimensions
50	FIELD overflow
51	Internal error
52	Bad file number
53	File not found
54	Bad file mode
55	File already open
57	Disk I/O error
58	File already exists
61	Disk full
62	Subscript out of range
63	Bad record number
64	Bad file name
66	Direct statement in file
67	Too many files
68	Internal error
69	Volume name not found
70	Rename error
71	Volume number error
72	Volume not enabled
73	Invalid Password
74	Illegal disk change
75	Write protected

76	Error in Parameter
77	Too many parameters
78	File not open
100	File/Record locked
101	Device not ready
102	Record not found
103	Record already exists
104	Key not found
105	Key already exists
106	Bad indexes
107	Invalid file operation
108	No primary index
109	Duplicate key

APPENDIX B. FUNCTION-KEY RETURNED VALUES

This appendix provides a list of function-keys associated to the corresponding value.

Function Key	Returned Value
(*) P5	(*) 129
(*) P4	(*) 130
(*) P3	(*) 131
(*) P2	(*) 132
(*) P1	(*) 133
P5	(*) 134
P4	(*) 135
P3	(*) 136
P2	(*) 137
P1	(*) 138
(*) S5	(*) 139
(*) S4	(*) 140
(*) S3	(*) 141
(*) S2	(*) 142
(*) EXIT	(*) 143
(*) ↑	(*) 144
(*) ↓	(*) 145
(*) →	(*) 146
(*) ←	(*) 147
(*) F16	(*) 148
(*) F15	(*) 149
(*) F14	(*) 150
(*) F13	(*) 151
(*) F12	(*) 152
(*) F11	(*) 153

(*)	F10	(*)	154
(*)	F9	(*)	155
	F8	(*)	156
	F7	(*)	157
	F6	(*)	158
	F5	(*)	159
	F4	(*)	160
	F3	(*)	161
	F2	(*)	162
	F1	(*)	163
	S5	(*)	164
	S4	(*)	165
	S3	(*)	166
	S2	(*)	167
	EXIT	(*)	168
(*)	DL	(*)	169
(*)	IL	(*)	170
(*)	HC	(*)	171
(*)	CH-WIND	(*)	172
	SEND	(*)	173
	SKIP	(*)	174
(*)	DC	(*)	175
(*)	IC	(*)	176
(*)	CL.ERR	(*)	177
	↑	(*)	178
	↓	(*)	179
	→	(*)	180

	←	(*)	181
	→|	(*)	182
	|←	(*)	183
(*)	CLEAR	(*)	185
	HOME	(*)	186
	ERASE	(*)	187

(*) With CONTROL key pressed.

APPENDIX C. ASCII CODES

This appendix shows decimal, hexadecimal, and binary representation of the ASCII code.

a	b	c	d	a	b	c	d	a	b	c	a	b	c
0	00	0000 0000	NUL	64	40	0100 0000	@	128	80	1000 0000	192	C0	1100 0000
1	01	0000 0001	SOH	65	41	0100 0001	A	129	81	1000 0001	193	C1	1100 0001
2	02	0000 0010	STX	66	42	0100 0010	B	130	82	1000 0010	194	C2	1100 0010
3	03	0000 0011	ETX	67	43	0100 0011	C	131	83	1000 0011	195	C3	1100 0011
4	04	0000 0100	EQT	68	44	0100 0100	D	132	84	1000 0100	196	C4	1100 0100
5	05	0000 0101	ENQ	69	45	0100 0101	E	133	85	1000 0101	197	C5	1100 0101
6	06	0000 0110	ACK	70	46	0100 0110	F	134	86	1000 0110	198	C6	1100 0110
7	07	0000 0111	BEL	71	47	0100 0111	G	135	87	1000 0111	199	C7	1100 0111
8	08	0000 1000	BS	72	48	0100 1000	H	136	88	1000 1000	200	C8	1100 1000
9	09	0000 1001	HT	73	49	0100 1001	I	137	89	1000 1001	201	C9	1100 1001
10	0A	0000 1010	LF	74	4A	0100 1010	J	138	8A	1000 1010	202	CA	1100 1010
11	0B	0000 1011	VT	75	4B	0100 1011	K	139	8B	1000 1011	203	CB	1100 1011
12	0C	0000 1100	FF	76	4C	0100 1100	L	140	8C	1000 1100	204	CC	1100 1100
13	0D	0000 1101	CR	77	4D	0100 1101	M	141	8D	1000 1101	205	CD	1100 1101
14	0E	0000 1110	SO	78	4E	0100 1110	N	142	8E	1000 1110	206	CE	1100 1110
15	0F	0000 1111	SI	79	4F	0100 1111	O	143	8F	1000 1111	207	CF	1100 1111
16	10	0001 0000	DLE	80	50	0101 0000	P	144	90	1001 0000	208	D0	1101 0000
17	11	0001 0001	DC_1	81	51	0101 0001	Q	145	91	1001 0001	209	D1	1101 0001
18	12	0001 0010	DC_2	82	52	0101 0010	R	146	92	1001 0010	210	D2	1101 0010
19	13	0001 0011	DC_3	83	53	0101 0011	S	147	93	1001 0011	211	D3	1101 0011
20	14	0001 0100	DC_4	84	54	0101 0100	T	148	94	1001 0100	212	D4	1101 0100
21	15	0001 0101	NAK	85	55	0101 0101	U	149	95	1001 0101	213	D5	1101 0101
22	16	0001 0110	SYN	86	56	0101 0110	V	150	96	1001 0110	214	D6	1101 0110
23	17	0001 0111	ETB	87	57	0101 0111	W	151	97	1001 0111	215	D7	1101 0111
24	18	0001 1000	CAN	88	58	0101 1000	X	152	98	1001 1000	216	D8	1101 1000
25	19	0001 1001	EM	89	59	0101 1001	Y	153	99	1001 1001	217	D9	1101 1001
26	1A	0001 1010	SUB	90	5A	0101 1010	Z	154	9A	1001 1010	218	DA	1101 1010
27	1B	0001 1011	ESC	91	5B	0101 1011	[	155	9B	1001 1011	219	DB	1101 1011
28	1C	0001 1100	FS	92	5C	0101 1100	\	156	9C	1001 1100	220	DC	1101 1100
29	1D	0001 1101	GS	93	5D	0101 1101	]	157	9D	1001 1101	221	DD	1101 1101
30	1E	0001 1110	RS	94	5E	0101 1110	↑	158	9E	1001 1110	222	DE	1101 1110
31	1F	0001 1111	US	95	5F	0101 1111	—	159	9F	1001 1111	223	DF	1101 1111
32	20	0010 0000	SPACE	96	60	0110 0000	`	160	A0	1010 0000	224	E0	1110 0000
33	21	0010 0001	!	97	61	0110 0001	a	161	A1	1010 0001	225	E1	1110 0001
34	22	0010 0010	"	98	62	0110 0010	b	162	A2	1010 0010	226	E2	1110 0010
35	23	0010 0011	#	99	63	0110 0011	c	163	A3	1010 0011	227	E3	1110 0011
36	24	0010 0100	$	100	64	0110 0100	d	164	A4	1010 0100	228	E4	1110 0100
37	25	0010 0101	%	101	65	0110 0101	e	165	A5	1010 0101	229	E5	1110 0101
38	26	0010 0110	&	102	66	0110 0110	f	166	A6	1010 0110	230	E6	1110 0110
39	27	0010 0111	'	103	67	0110 0111	g	167	A7	1010 0111	231	E7	1110 0111
40	28	0010 1000	(	104	68	0110 1000	h	168	A8	1010 1000	232	E8	1110 1000
41	29	0010 1001	)	105	69	0110 1001	i	169	A9	1010 1001	233	E9	1110 1001
42	2A	0010 1010	*	106	6A	0110 1010	j	170	AA	1010 1010	234	EA	1110 1010
43	2B	0010 1011	+	107	6B	0110 1011	k	171	AB	1010 1011	235	EB	1110 1011
44	2C	0010 1100	,	108	6C	0110 1100	l	172	AC	1010 1100	236	EC	1110 1100
45	2D	0010 1101	-	109	6D	0110 1101	m	173	AD	1010 1101	237	ED	1110 1101
46	2E	0010 1110	.	110	6E	0110 1110	n	174	AE	1010 1110	238	EE	1110 1110
47	2F	0010 1111	/	111	6F	0110 1111	o	175	AF	1010 1111	239	EF	1110 1111
48	30	0011 0000	0	112	70	0111 0000	p	176	B0	1011 0000	240	F0	1111 0000
49	31	0011 0001	1	113	71	0111 0001	q	177	B1	1011 0001	241	F1	1111 0001
50	32	0011 0010	2	114	72	0111 0010	r	178	B2	1011 0010	242	F2	1111 0010
51	33	0011 0011	3	115	73	0111 0011	s	179	B3	1011 0011	243	F3	1111 0011
52	34	0011 0100	4	116	74	0111 0100	t	180	B4	1011 0100	244	F4	1111 0100
53	35	0011 0101	5	117	75	0111 0101	u	181	B5	1011 0101	245	F5	1111 0101
54	36	0011 0110	6	118	76	0111 0110	v	182	B6	1011 0110	246	F6	1111 0110
55	37	0011 0111	7	119	77	0111 0111	w	183	B7	1011 0111	247	F7	1111 0111
56	38	0011 1000	8	120	78	0111 1000	x	184	B8	1011 1000	248	F8	1111 1000
57	39	0011 1001	9	121	79	0111 1001	y	185	B9	1011 1001	249	F9	1111 1001
58	3A	0011 1010	:	122	7A	0111 1010	z	186	BA	1011 1010	250	FA	1111 1010
59	3B	0011 1011	;	123	7B	0111 1011	{	187	BB	1011 1011	251	FB	1111 1011
60	3C	0011 1100	<	124	7C	0111 1100	|	188	BC	1011 1100	252	FC	1111 1100
61	3D	0011 1101	=	125	7D	0111 1101	}	189	BD	1011 1101	253	FD	1111 1101
62	3E	0011 1110	>	126	7E	0111 1110	~	190	BE	1011 1110	254	FE	1111 1110
63	3F	0011 1111	?	127	7F	0111 1111	DEL	191	BF	1011 1111	255	FF	1111 1111

Note Boxed characters are different on national keyboards.

Code 4002340 B (0)
Printed in Italy