Enterprise IS-BASIC v3.0 - Specification

22 April 1985

CONTENTS

1. Introduction

2. Formal Specification

   2.1 Programs
   2.2 Expressions
   2.3 Variables
   2.4 Identifiers
   2.5 Constants
   2.6 Characters

3. Expressions and Arithmetic

   3.1 Numbers and Strings
   3.2 Operators and Precidence

4. General Rules of Use

5. Commands and Statements

6. Functions

7. Errors

8. Index

1. INTRODUCTION

This document describes IS-BASIC version 3.0 as implemented on the Enterprise under the EXOS operating system.

Version 3 and above includes many small enhancements over the previous versions 2.0 and 2.1 .

IS-BASIC is an interpreted BASIC based on the proposed ANSI Standard X3J2/82-17 and is compatible with the ISO 6373 Standard for Minimal BASIC which is the proposed BS 6373 Standard for Minimal BASIC. The OPTION BASE construct which is one of the two UK objections to the adoption of this Standard as a British Standard is not provided, being made somewhat redundant by the more flexible syntax of the ANSI Standard.

IS-BASIC provides a command loop from which programs and commands can be entered via the keyboard. Many facilities are provided for the editing of programs such as the AUTO, RENUMBER and DELETE. The ability exists for there to be more than one program in memory at any one time, and from immediate mode any program may be made the current program so that it may be edited or run independantly of the other programs currently in memory. A running program may invoke another program by name that is either currently in memory or is on cassette / disk. The passing of parameters by value is supported during this program chaining.

This is an extension of the various statements in IS-BASIC which enable the structured programming constructs required of modern programming languages to be used, such as repeat-until and do-while loops (implemented in IS-BASIC with DO WHILE/UNTIL ...LOOP WHILE/UNTIL loops), CASE selections and multi-line IF...ELSE...THEN constructs.

More traditional BASIC statements such as GOTO, GOSUB and ON..GOTO are provided for compatibility.

This document is arranged in two main parts split over several chapters. The two main parts are:

- Formal BNF specification of the primary elements of IS-BASIC (chapters 0 to 2)
- A more readable and general description of the statements, commands and functions available from IS-BASIC (chapters 3 onwards)

2. FORMAL BNF SYNTAX SPECIFICATION

This chapter gives a formal syntactic specification of IS-BASIC in a modified form of BNF. The semantic action associated with each syntactic construct is specified elsewhere.

Note that certain (context dependant) syntactic constructs that are in fact not allowed or are meaningless are not ruled out by the notation used (described below), but are mentioned in the more general descriptions elsewhere (eg. Chapter 5 'Commands and Statements'). Similarly, certain limits are also not specified by the notation. These are also described in the more general descriptions or are limited by other factors such as the maximum length of program lines or the amount of memory available.

BNF items in lower-case are "metanames" ie. the names of syntactic objects, and are defined in terms of other metanames. To prevent this process recursing indefinitely, certain metanames are 'terminal' metanames and are generally ASCII characters or strings delimited by single quotes. Textual comments are given between brackets ([and]).

The mataname '?' indicates that the preceding object is optional, the metaname '*' indicates that the preceding object cam be repeated zero or more times, and the metaname '/' indicates that either the the preceding or following object may be given. The precedence of ?, * and / is the same, and ? and * group from left to right and / groups from right to left. The precedence may be overriden with paranthesis ('(' and ')').

2.1 Programs

2.1.1 General description

A BASIC program constant of a number of lines each preceded by a line-number which serves a label for the line to facilitate editing and as a label for the first statement on the line for certain statements and exception-handler related functions.

2.1.2 Syntax

```
program                 = program-name-line? program-line*
program-name-line       = line-number 'PROGRAM' program-
                          name parameter-list? tail
line-number             = digit digit? digit? digit?
program-name            = quoted-string / unquoted-string-
                          character
tail                    = tail-comment? end-of-line
tail comment            = '!' remark-string
remark-string           = character*
end-of-line             = [ ASCII carraiage return ]
program-line            = line / statement-line
line                    = case-line / data-line / def-
                          line / dim-line / do-line / else-
                          line / else-if-line / end-line /
                          end-def-line / end-handler-line /
                          end-if-line / end-select-line /
                          end-when-line / for-line /
                          handler-line / image-line / if-
                          line / look-line / loop-line /
                          next-line / numeric-line / on-
                          line / option-line / rem-line /
                          retry-line / select-line /
                          string-line / when-line
statement-line          = line-number statements? tail
statements              = statement (':' statement)*
statement               = allocate-statement / ask-
                          statement / call-statement /
                          capture-statement / cause-
                          statement / chain-statement /
                          clear-statement / close-
                          statement / code-statement /
                          continue-statement / copy-
                          statement / date-statement /
                          display-statement / envelope-
                          statement / exit-def-statement /
                          exit-do-statement / exit-for-
                          statement / exit-handler-
                          statement / ext-statement /
                          flush-statement / get-statement /
                          gosub-statement / goto-
                          statement / graphics-statement /
```


input-statement / let-statement / line-input-statement / lprint-statement / open-statement / out-statement / ping-statement / plot-statement / poke-statement / print-statement / randomize-statement / read-statement / redirect-statement / restore-statement / return-statement / run-statement / set-statement / sound-statement / spoke-statement / stop-statement / text-statement / time-statement / toggle-statement / trace-statement / type-statement / wait-statement

command = allocate-statement / ask-statement / auto-command / call-statement / capture-statement / cause-statement / clear-statement / close-statement / code-statement / continue-statement / copy-statement / date-statement / delete-command / display-statement / chain-statement / edit-command / envelope-statement / ext-statement / flush-statement / get-statement / graphics-statement / info-command / let-statement / list-command / llist-command / load-command / look-line / lprint-statement / merge-command / new-command / ok-command / open-statement / option-line / out-statement / ping-statement / plot-statement / poke-statement / print-statement / redirect-statement / rem-statement / renumber-command / run-statement / save-command / set-statement / sound-statement / spoke-statement / start-command / text-statement / time-statement / toggle-statement / trace-statement / type-statement / verify-command / wait-statement

A line-number of zero is not allowed, and leading zeros have no effect.

The maximum length of a line is 250 characters, and is terminated by an ASCII carriage return. Characters between the 250th character and the carriage return are ignored.

2.1.3 Semantics

The program-name-line allows the program to be refered to by name and may also serve as a comment. It forms the operand of the chain-statement and may also be used in the save-command. In this case, only the quoted-string form of program-name is used; if the other form is used then the effect obtained when using chain-statements and save-commands is that obtained when there is no program-name-line ie. the program cannot be refered to by name. Similarly a parameter-list given in a program-name-line is only recognised when the program-name is a quoted-string. A program-name-line may occur anywhere in a program but will only serve to name the program if it appears on the first line of the line of the program. In any case the effect of executing a program-name-line is to cause execution to continue to the next line of the program.

The parameters to the program specified in the parameter-list in the program-name-line are passed by value. An exception will result if the parameters being passed are of the wrong type, but no exception will occur if the number of parameters do not match. Extra parameters in the chain-statement or run-command are ignored, and extra parameters in the program-name-line are set to zero for numeric parameters and the null string for string parameters.

Lines in the program are executed in sequential order according to their line numbers until:

- some other action is specified by the execution of a program-line
- an execption occurs and a user-defined exception handler is active
- execution of the program ends as a result of executing an end-statement, a stop-statement, a chain-statement or a run-statement
- execution of the program ends as a result of the STOP key being pressed.
- execution of the program ends by there being no more statements or lines to execute in the sequential order of the program.

2.1.4 Examples

```
10 PROGRAM "myprog" (A,B,C$)
20 IF A=B THEN                          ! no error
30   PRINT "OK": PRINT: PRINT
40 END IF

100 PROGRAM TEST                        ! not chained to
```

2.1.5 Remarks

A command may be entered for immediate execution. This is called 'immediate mode', and in this case no line-number is given. Program execution starts when in immediate mode a command that has this effect is typed (eg. the run-statement). After execution of the program has ended, immediate mode is returned to.

2.2 Expressions

2.2.1 General description

Expressions are obtained by applying the primary operations associated with each type (addition, subtraction, multiplication, division and involution for numeric expressions and concatenation and slicing for string expressions) to values obtained from constants, simple-variables, array-elements and functions-refs. Since the latter two may also invoke the expression evaluating mechanism, the process is recursive.

2.2.2 Syntax

```
expression                = numeric-expression / string-
                            expression
numeric-expression        = numeric-disjunction
numeric-disjunction       = logical-disjunction / bitwise-
                            disjunction
logical-disjunction       = numeric-conjunction ('OR'
                            numeric-conjunction)*
bitwise-disjunction       = numeric-conjunction ('BOR'
                            numeric-conjunction)*
numeric-conjunction       = logical-conjunction / bitwise-
                            conjunction
logical-conjunction       = comparison ('AND' comparison)*
bitwise-conjunction       = comparison ('BAND' comparison)*
comparison                = 'NOT'? (numeric-comparison /
                            string-comparison)
numeric-comparison        = arithmetic-expression (logical-
                            operator arithmetic-expression)*
string-comparison         = string-expression logical-
                            operator string-expression
logical-operator          = '<' / '=' / '>' / '<=' / '=<' /
                            '>=' / '=>'
arithmetic-expression     = sign? arithmetic-term ( ('+' /
                            '-') arithmetic-term)*
arithmetic-term           = factor ( ('*' / '/') factor)*
factor                    = numeric-primary ('^' numeric-
                            primary)*
numeric-primary           = numeric-rep / numeric-variable /
                            numeric-function-ref / ( '('
                            numeric-expression ')' )
numeric-function-ref      = numeric-function argument-list?
numeric-function          = numeric-supplied-function /
                            numeric-defined-function
numeric-supplied-function= abs-call / acos-call / angle-
                            call / asin-call / atn-call /
                            bin-call / black-call / blue-
                            call / ceil-call / cos-call /
                            cosh-call / cot-call / csc-call /
                            cyan-call / deg-call / eps-call /
```



```
                                exline-call / exp-call / extype-
                                call / free-call / fp-call /
                                green-call / in-call / inf-call /
                                int-call / ip-call / joy-call /
                                lbound-call / len-call / log-
                                call / log2-call / log10-call /
                                majenta-call / max-call / maxlen-
                                call / min-call / mod-call / ord-
                                call / peek-call / speek-call /
                                pi-call / pos-call / rad-call /
                                red-call / rem-call / rgb-call /
                                rnd-call / round-call / sec-
                                call / sin-call / sinh-call /
                                size-call / sgn-call / sqr-call /
                                tan-call / tanh-call / truncate-
                                call / ubound-call / usr-call /
                                val-call / vernum-call / white-
                                call / yellow-call
numeric-defined-function = numeric-identifier
argument-list            = argument (',' argument)*
argument                 = value-argument / reference-
                           argument
value-argument           = expression
referenece-argument      = variable
string-expression        = substring ('&' substring)*
substring                = string-primary ('(' substring-
                           specifier ')')*
substring-specifier      = left-substring / right-substring
left-substring           = numeric-expression right-
                           substring?
right-substring          = ':' numeric-expression?
string-primary           = string-constant / string-
                           variable / string-function-ref
string-function-ref      = string-function argument-list?
string-function          = string-supplied-function /
                           string-defined-function
string-supplied-function = chr-call / date-call / hex-call /
                           inkey-call / lcase-call / ltrim-
                           call / rtrim-call / str-call /
                           time-call / ucase-call / ver-
                           call / word-call
string-definied-function = string-identifier
```

2.2.3 Semantics

In a numeric-expression, the symbols '^', '*', '/', '+' and '-' represent the operations of involution, multiplication, division, addition and subtraction or negation respectively. Unless paranthesis dictate otherwise, the order of evaluation is involution, multiplication and division, and then addition, subtraction and negation. In the absence of paranthesis, the order of evaluation of operators of equal precedence is left to right.

If an underflow occurs during the evaluation of a numeric-expression, then the value which generated the underflow will be replaced by zero.

0^0 is defined to be 1.

A numeric-function-ref represents the value obtained as a result of the invocation of the function, after first evaluating the arguments and substituting them for the parameters specified in the function-definition (for numeric-defined-functions) or the definition (for numeric-supplied-functions).

The operators 'AND' and 'OR' both return a truth value of zero (for false) or minus one (for true), and assume that a non-zero operand represents true. The evaluation of logical-disjunctions and logical-conjunctions guarantees that certain parts of the expression will not be evaluated if it is not necessary to do so in order to determine the truth value of the final result for that logical-disjunction or logical-conjunction.

The elements of a logical-disjunction are evaluated from left to right until a truth value of 'true' is obtained; the result of the logical-disjunction is then known and the remainder is not evaluated. If none of the elements of a logical-disjunction evaluate to true, then the result is false.

The elements of a logical-conjunction are evaluated from left to right until a truth value of 'false' is obtained; the result of the logical-conjunction is then known and the remainder is not evaluated. If none of the elements of a logical-conjunction evaluate to false, then the result is true.

The operator 'NOT' reverses the truth value of its operand ie. a non-zero operand is replaced by zero, and an operand with a value of zero is replaced by minum one.

The operators 'BAND' and 'BOR' perform a bitwise AND and OR respectively. Their operands must be integers in the range -9999 to 9999 for the expected result to be obtained.

The operators '<=' and '=<' (not greater than) are equivalent, as are the operators '>=' and '=>' (not less than). The operators '<', '=' and '>' represent less than, equal to and greater than respectively.

The result of a numeric-comparison is a truth value obtained as a result of the algebraic comparison between the two operands according to the operator.

The result of a string-comparison is a truth value obtained as a result of the comparison between the two strings according to the operator. Two strings are 'equal' if they have the same length and contain an identical sequence of characters. The relation between two strings is determined by comparing them character-by-character for as many characters as are contained in the shorter string (which may be zero) or until the characters compared do not have the same ASCII value. In the former case (when one string is the initial left sub-string of the other) the shorter string is 'less than' the longer one. In the latter case, the string in which the leftmost character position in which the strings differ precedes the corresponding character in the other string according to its ASCII value is 'less than' the other string.

The accuracy to which numeric-expressions are evaluated is 12 decimal digits internally, yielding a result of 10 decimal digits for displaying or assigning to a numeric-variable. Numeric-supplied-functions also work to an accuracy of 12 decimal digits internally.

A substring-specifier represents a portion of a given string from the character position indicated by the left-substring (if given) to the character position indicated by the right-substring (if given). The left-substring defaults to the first character position in the string - 1, and the right-substring defaults to the last character position in the string + 1. Character positions are numbered incrementally from 1, which represents the first character in the string. A left-substring of less than 1 indicates the first character of the string, and a right-substring of greater than the length of the string represents the last character of the string. If the right-substring is less than the left-substring, then a null string is indicated. If no right-substring is specified in the substring-specifier, then the single character indicated by the left-substring is specified.

The range of values that can be represented during the calculation of a numeric-expression is -0.999999999999E+63 to 0.999999999999E+63. Values outside this range constitute an overflow and result in an exception being caused. Values between -0.1E-63 and 0.1E-63 constitute an underflow, and are replaced by zero.

The result of a numeric-expression is rounded from 12 to 10 digits after computation, and if the rounded value lies outside the range -0.9999999999E+63 to 0.9999999999E+63, then an overflow has occured and an exception results.

The maximum length of a string during and as a result of a string-expression is 254 characters. The characters may have any ASCII value between 0 and 255 inclusive.

2.2.4 Examples

```
A
1*5
-X^Y                    ! equivalent to -(X^Y)
2^(-X)
SIN(2*TAN(X))
If X(y) is an array with elements from 1 to 10, then:
     Y>=1 AND Y>=10 AND X(Y)=2
will never cause an 'out of bounds' exception
A$
A$ & "string"
"string"(2:4)           ! equivalent to "tri"
"string"(4:2)           ! equivalent to ""
"string"(:4)            ! equivalent to "stri"
"string"(2:)            ! equivalent to "tring"
"string"(4)             ! equivalent to "i"
"string"(:)             ! equivalent to "string"
"string"(2:5)(:3)       ! equivalent to "trin"(:3) = "tri"
FRED$(10:20)(X:Y)
```

2.2.5 Remarks

It should be noted that the unary negation operator '-' has the same precedence as the binary subtraction operator '-'. This differs from many BASIC interpreters, in which unary minus has the highest precedence of all operators. Also the involution operator '^' is evaluated from left to right, which is in the oppposite direction to the convention used by mathematicians and some pocket calculators.

2.3 Variables

2.3.1 General description

Variables may be either simple variables, which reference one value, or may be references to an element of a one or more dimensional array. Variables contain data of the appropriate type which can be modified by the program.

2.3.2 Syntax

```
variable                = numeric-variable / string-
                          variable
numeric-variable        = simple-numeric-variable /
                          numeric-array-element
simple-numeric-variable = numeric-identifier
numeric-array-element   = numeric-identifier subscript-part
subscript-part          = '(' subscript (',' subscript)?
                          ')'
subscript               = index
index                   = numeric-expression
string-variable         = simple-string-variable / string-
                          array-element
simple-string-variable  = string-identifier
string-array-element    = string-identifier subscript-part
simple-variable         = simple-numeric-variable / simple-
                          string-variable
array-element           = numeric-array-element / string-
                          array-element
```

2.3.3 Semantics

Simple-variables are declared implicitly through their appearance in a program, and their life-time (ie. scope) depends upon the history of execution of program lines, and is dynamic.

Array-elements are declared explicitly in a dim-statement, numeric-statement or string-statement, and subscripts must have values in the declared range, and the number of subscripts must equal the number of declared dimensions.

At the initiation of the execution of the program all values associated with variables will be undefined, and an exception will result if a variable is used in a context where its value is required when no value has yet been assigned to it during execution of the program. An exception to this is variables declared as parameters to a program. These are initialised as described in section 2.1.3.

2.3.4 Examples

```
X
MY_VAR(4,5)
A$(100)
```

2.3.5 Remarks

2.4 Identifiers

2.4.1 General description

Identifiers are used to name variables, arrays, functions and programs.

2.4.2 Syntax

```
identifier              = numeric-identifier / string-
                          identifier
numeric-identifier      = letter identifier-character*
string-identifier       = numeric-identifier '$'
```

Identifiers cannot contain more than 31 characters (including the '$' at the end of a string-identifier).

No identifier can name more than one object or type of object, although the numeric-identifier part of a string-identifier may be the same as a numeric-identifier ie. two different types of identifier may contain a sequence of characters differing only in the final '$' of the string-identifier.

2.4.3 Semantics

Identifiers declared (either explicitly or by their first use during execution of the program) outside a function-definition are global to the entire program. Identifiers declared within a function definition are local to each execution of the function and to any functions that are invoked by that function, unless that function declares an identifier of the same name. That is, the scope of identifiers within a program is dynamic, and at any one time during the execution of the program the identifiers which have been declared depends upon the history of which lines have been executed.

2.4.4 Examples

```
X
A$
MY_VAR
El0
LAST.CHAR.USED
```


2.4.5 Remarks

Any identifier may be declared which has the same name as a statement, line, command or supplied-function. If an identifier has the same name as a supplied-function, then that supplied-functions is overriden by the declared identifier. If an identifier has the same name as a statement, line or command, then use of that statement, line or command is not affected by the identifier.

2.5 Constants

2.5.1 General description

Constants represent either a fixed numeric value or a fixed string of characters. The two types of constant (numeric-constant and string-constant) are the two primative data types in IS-BASIC (ie. numbers and strings).

2.5.2 Syntax

```
constant          = numeric-constant / string-
                    constant
numeric-constant  = sign? numeric-rep
sign              = '+' / '-'
numeric-rep       = mantissa exponent?
mantissa          = (integer '.'?) / (integer?
                    fraction)?
integer           = digit digit*
fraction          = '.' integer?
exponent          = 'E' sign? integer
string-constant   = quoted-string
quoted-string     = '"' quoted-string-character* '"'
```

2.5.3 Semantics

Where a sign is optional and no sign is given, a '+' is assumed. The value of a numeric-constant is the value represented by the mantissa multiplied by ten raised to the power of the value represented by the exponent.

Numeric-constants may have an arbitrary number of digits, although extra digits in the mantissa will be truncated after the tenth significant digit. The range of numeric-constants is -0.9999999999E+63 to 0.9999999999E+63. If the value of a numeric-constant lies in the range -0.1000000000E-63 to 0.1000000000E-63 exclusive, then a value of zero is assumed (as for numeric underflow).

The value of a string-constant is the sequance of all the characters between the initial and final '"'. Spaces are significant and upper- and lower-case are distinct.

A single '"' may be represented in a string-constant by '""'.

The maximum length of a string-constant is limited by the maximum length of a program-line.

2.5.4 Examples

```
1
0.1
.123
.1E34
-.1223000E-0012
"FRED"
"0.1"
"He said, ""IS-BASIC""."
""""
```

2.5.5 Remarks

2.6 Characters

2.6.1 General description

On input, all ASCII printing characters are valid, but lower-case characters are converted to their upper-case equivalents except for those appearing inside quoted-strings, unquoted-strings or remark-strings. Control characters (below ASCII space) are ignored. The top bit of all characters is reset.

2.6.2 Syntax

```
character                 = '"' / quoted-string-character
quoted-string-character   = '!'/'"'/'£'/'$'/'%'/'&'/'''/'('/
                            ')'/'*'/','/','/'/'/':'/';'/'<'/
                            '='/'>'/'?'/'@'/'['/'\','']'/'^'/
                            '_'/'`'/'{'/'|'/'}'/'~'/
                            unquoted-string-character
unquoted-string-character= ' ' / plain-string-character
plain-string-character    = digit / letter / '.' / '+' / '-'
digit                     = '0'/'1'/'2'/'3'/'4'/'5'/'6'/'7'/
                            '8'/'9'
letter                    = 'A'/'B'/'C'/'D'/'E'/'F'/'G'/'H'/
                            'I'/'J'/'K'/'L'/'M'/'N'/'O'/'P'/
                            'Q'/'R'/'S'/'T'/'U'/'V'/'W'/'X'/
                            'Y'/'Z'/'a'/'b'/'c'/'d'/'e'/'f'/
                            'g'/'h'/'i'/'j'/'k'/'l'/'m'/'n'/
                            'o'/'p'/'q'/'r'/'s'/'t'/'u'/'v'/
                            'w'/'x'/'y'/'z'
identifier-character      = letter / digit / '_' / '.'
```

2.6.3 Semantics

2.6.4 Examples

2.6.5 Remarks

The syntax as described produces a program consisting of no spaces except in quoted-strings, unquoted-strings and remark-strings. Spaces may however occure anywhere in a program-line except in the following cases:

- within a line-number
- within numeric-constants
- within identifiers
- within keywords
- within multi-character relational symbols

EXPRESSIONS AND ARITHMETIC

Chapter 2 specifies the syntax of expressions and describes how they are evaluated. This chapter covers some more general points about expressions and arithmetic, and describes them in a more general way.

3.1 Numbers and Strings

All arithmetic is performed internally to 12 decimal digits of accuracy and for the purposes of printing and storing in users variables these 12-digit numbers are rounded to 10 digits. The exponent can be ±63 with a leading decimal point on the mantissa. Chapter 2 section 2 describes more fully the full range of numbers.

Strings may be 0 to 254 characters in length, and may contain any character.

A fixed amount of RAM is allocted to each string, and can be defined with the statement STRING (see the definition of string-line in chapter 5). The default maximum length of strings is 132 characters, and if string arrays are used, it is particularly useful to dimension the elements to less than this since the amount of memory thus saved is (total number of elements) * (132-new maximum length), which is often quite significant.

3.2 Operators

The numeric operators available, in order of precedence (highest first), are:

^	- Involution. 0^0 is 1. 0^x is 0. x^0 is 1. An attempt to raise a negative number to a non-integral power results in an error, as does an attempt to raise zero to a negative power.
*, /	- Multiplication and division. An attempt to divide by zero results in an error.
+, -	- Addition, subtraction and unary negation.
=, <>, <, <=, >, >=	- Relational tests for equal, not equal, less than, not greater than, greater than and not less than Alternative forms for <= and >= are =< and => respectively.
NOT	- Logical complement.

BAND, AND - Bitwise and logical AND.
Logical AND will only evaluate a sub-expression as far as is necessary to determine the result.

BOR, OR - Bitwise and logical OR.
Logical OR will only evaluate a sub-expression as far as is necessary to determine the result.

Note that binary subtraction and unary minus have the same precedence for compaibility with ANSI and Minimal BASIC.

The string operators available, in order of precedence (highest first), are:

(x:y) - Slicing
Returns the string from the xth character to the yth character. If x or y are missing, or if x<1 or y>LEN(string), then the default values of the first character and the last character are assumed.

& - Concatenation
The result must be less than 254 characters in length.

3.3 Examples

```
PRINT 1+2*3^(6-2)          Answer 163
PRINT 2 BOR 8              Answer 10
PRINT -2 BAND 11           Answer 10 (equivalent to FFFE
                           AND 000B)
PRINT "FRED"(-3:3)         Answer "FRE"
PRINT "FRED"(2:)           Answer "RED"
PRINT "FRED"(:1000)        Answer "FRED"
PRINT "FRED"(3)            Answer "E"
PRINT "FRED"(5:3)          Answer ""
PRINT F$(K:K+4)(2:3)       Equivalent to F$(K:1:K+2)
```


GENERAL RULES OF BASIC

Program lines are inserted into the program via the keyboard by typing in the line preceded by a line number in the range 1 to 9999. Any existing lines with the same line number are deleted. Typing in the line number alone just deletes any existing line with the same number from the program. Typing in the line number followed by a space inserts a comment line.

Statements may be separated on a line with a colon (':').

Spaces may be put anywhere on the line except in keywords, multiple-character relational signs (<= etc.) and identifiers. Spaces must separate keywords, identifiers and expressions.

Each line can have a maximum length of 250 characters and is terminated with a carriage return (Enter key). Characters between the 250th and the carriage return are ignored. All letters are uppercased except those in quoted-strings, unquoted-strings, remark-strings and data-statements.

5. STATEMENTS AND COMMANDS

This chapter defines the syntax and semantics of the statements, lines and commands used in the definition of a program in chapter 2.

Statements are those BASIC keywords which when executed in a BASIC program do some specific action. They frequently call the expression evaluation mechanism which in turn calls the arithmetic routines which operate on constants, variables and functions. User-defined functions can contain any statement.

Lines are those statements which require a program line of their own, and often start or end an indented program block. Statements by contrast can share a program line, and are separated by a colon (':'). This is called a multi-statement line.

Commands are those statements which can only be used outside a program (ie. in 'immediate mode'), and are generally concerned with the editing of programs. Many (but not all) statements can also be used in immediate mode. Lines cannot be used in immediate mode, since they generally require a corresponding line elsewhere in the program (for example a line which starts a block requires the block end to be elsewhere in the program).

Each statement, command or line is described in four parts:

- Syntax definition
- Semantics and description
- Examples
- Associated keywords

ALLOCATE

Syntax

```
allocate-statement      = 'ALLOCATE' numeric-expression
```

Description

The allocate-statement is used to obtain RAM for use by the BASIC programmer, and is generally used to contain machine code which is later executed from the BASIC program. The numeric-expression is the number of bytes required. The start address of the memory thus obtained is not predictable, but can be obtained by use of the code-statement. The amount of memory that may be allocated depends upon other RAM usage by BASIC, but is typically 8 or 9 K. An exception results if there is insufficient RAM available.

The space is de-allocated when the program is run or when any other action which clears the symbol table is performed eg. when a program line is added or deleted or when a new program is loaded. The memory is not de-allocated when another program is made the current program, so that a program may allocate some space and put data there which may be used by another program.

Examples

```
ALLOCATE 200
```

Associated keywords

CODE, HEX$, WORD$. See code-statement, hex-call, word-call.

ASK

Syntax

```
ask-statement           = 'ASK' exos-variable numeric-
                          variable
exos-variable           = exos-variable-text / exos-
                          variable-number
exos-variable-text      = 'DEFAULT CHANNEL' / 'TIMER' /
                          'KEY CLICK' / 'KEY RATE' /
                          'KEY DELAY' / 'INTERRUPT STOP' /
                          'INTERRUPT KEY' /
                          'INTERRUPT NET' /
                          'INTERRUPT CODE' / 'TAPE SOUND' /
```



```
                                'TAPE PROTECT' / 'TAPE LEVEL' /
                                'FAST SAVE' / 'SOUND STOP' /
                                'SOUND BUFFER' / 'SPEAKER' /
                                'SERIAL BAUD' / 'SERIAL FORMAT' /
                                'VIDEO MODE' / 'VIDEO COLOR' /
                                'VIDEO COLOUR' / 'VIDEO X' /
                                'VIDEO Y' / 'BORDER' / 'BIAS' /
                                'STATUS' / 'EDITOR TEXT' /
                                'EDITOR KEY' / 'EDITOR BUFFER' /
                                'REM1' / 'REM2' / 'NET CHANNEL' /
                                'NET NUMBER' / 'NET MACHINE' /
                                'SPRITE'
exos-variable-number        = numeric-expression
```

The exos-variable-text corresponds to the actual EXOS variables described in the EXOS documentation.

The exos-variable-number allows EXOS variables to be accessed by their number rather than by name, and may be useful for system extensions which implement new EXOS variables. The numeric-expression must be in the range 1 to 255.

Description

The ask-statement assigns the current value of some system parameter contained in an EXOS variable to the numeric-variable.

The exos-variable-number allows the EXOS variables to be accessed by number rather than by name, and may be useful for system extensions which implement new EXOS variables. The numeric-expression must evaluate to a value in the range 1 to 255.

Examples

```
ASK INTERRUPT CODE INT
ASK 78 A
```

Associated keywords

SET, TOGGLE. See set-statement, toggle-statement

AUTO

Syntax

```
auto-command                = 'AUTO' auto-option (',' auto-
                              option)*
```

```
auto-option          = at-number / step-size
at-number            = 'AT' line-number
step-size            = 'STEP' integer
```

The at-number if not specified defaults to line 100, and the step-size defaults to an increment of 10.

Description

The auto-command initiates automatic line number generation for convenience during the entering of a program from the keyboard. The line-number specified as the at-number is first printed, and then the program line (with no other line-numbers) can be entered. The step-size is then added to the at-number, and the new line-number thus obtained is printed to allow the next line to be entered. This process repeats until one of the following conditions is met:

- the new line-number exceeds 9999
- the STOP key is pressed
- an erroneous program line is entered
- a null line is entered (ie. Enter only pressed).

The at-number defaults to line 100, and the step-size defaults to 10. Any number of at-numbers or step-sizes may be given in the auto-options, but only the last one on the line will have an effect.

Examples

```
AUTO
AUTO AT 200 STEP 5
```

Associated keywords

None.

CALL

Syntax

```
call-statement       = 'CALL' numeric-expression
```


Description

The call-statement simply evaluates the numeric-expression and ignores the result. This is used to call machine code subroutines which do not return a result, and to call numeric-defined-functions, thus allowing user-defined functions to be used as procedures. In the latter case, the function definition does not have to assign a result to the function name. This would in other circumstances cause an 'undefined variable' exception.

Examples

```
CALL FRED(A, SIZE, G$)
CALL USR(MOVE)
```

Associated keywords

DEF, USR. See numeric-defined-functions, def-statement, usr-call.

CAPTURE

Syntax

```
capture-statement      = 'CAPTURE' source-channel?
                         destination-channel?
source-channel         = 'FROM' channel-number
destination            = 'TO' channel-number
channel-number         = '£' numeric-expression
```

The numeric-expression in channel can take values in the range 0 to 255.

Description

The capture-statement causes character reads from the destination-channel to come instead from the source-channel. The capture operation will cease when either of the following conditions occurs:

- The STOP key is pressed
- An error occurs from the destination-channel
- A subsequent capture-statement is performed with the same destination channel but with a source channel of 255.

The source-channel defaults to channel 0 (EDITOR:) and the destination-channel defaults to channel 104 (PRINTER:).

Examples

```
CAPTURE FROM £1 TO £105
```

Associated Keywords

COPY, REDIRECT. See copy-statement, redirect-statement.

CASE

Syntax

```
case-line           = line-number case-statement tail
case-statement      = 'CASE' case-list
case-list           = 'ELSE' / (case-item (',' case-
                      item)*)
case-item           = expression / range
range               = (expression 'TO' expression) /
                      ('IS' logical-operator
                      expression)
```

The expressions appearing in a case-statement must be of the same type as that in the corresponding select-line.

Case-lines must only appear inside a block beginning with a select-line and terminated by a end-select-line.

Description

A case-line matches with the expression in the corresponding select-line if one of the following conditions is true:

- the value of the expression in the select-line is equal to an expression in the case-item
- the value of the expression in the select-line is greater than or equal to that of the expression preceding 'TO' in the range, but less than or equal to the expression following the 'TO'.
- the value of the expression in the select line satisfies the relation indicated by a logical-operator after 'IS'.
- the case-list contains an 'ELSE'.

Examples

```
CASE 3,4,5
CASE IS < A$
CASE 2, 4 TO 5, 10
CASE ELSE
```

Associated keywords

SELECT, END SELECT. See select-line, end-select-line.

CAUSE

Syntax

cause-statement = 'CAUSE' 'EXCEPTION'? numeric-expression

Description

The cause-statement causes an exception with the number specified by the numeric-expression, which must be in the range 0 to 65535. This can either be used to simulate a system error, or to cause special errors that an exception handler can recognise. For the latter purposes, exceptions 0 to 999 should be used since future versions of BASIC will not use these, but may use higher numbered exception numbers.

Examples

```
CAUSE 52
```

Associated keywords

WHEN, HANDLER, EXTYPE, EXLINE. See when-line, handler-line, extype-call, exline-call.

CHAIN

Syntax

chain-statement = 'CHAIN' program-designator argument-list?

program-designator = string-primary / numeric-expression

Description

The chain-statement is used to transfer execution to another program that is currently in memory, possibly passing parameters by value to the chained to program.

If the program-designator is a numeric-expression, then the value of the expression is the number of the program in memory.

Note that the alternative program-designator is a string-primary. (A full string-expression could have a left paranthesis at the start of a string slice operation. This causes an abiguity with a paramter-list.) The string-primary must be equal to the name of the program given as a quoted-string in a program-line. Both strings are uppercased before being compared.

The total number of bytes taken up by the parameters must not exceed 256 bytes, and are passed as described in section 2.1 . Numerical parameters take up 9 bytes each, and strings take up the length of the string + 1.

Examples

```
CHAIN "END" (CHAN, NAME$)
CHAIN 4
```

Associated keywords

RUN, EDIT. See run-statement, edit-command, program-name.

CLEAR

Syntax

clear-statement = 'CLEAR' (channel ':')? (clear-item (';' clear-item)*)?

clear-item = 'NETWORK' / 'FKEYS' / 'FONT' / 'SCREEN' / 'TEXT' / 'GRAPHICS' / 'SOUND' / 'ENVELOPE' / ('QUEUE' numeric-expression)

If the channel is given in the clear-statement, then either a clear-item is not given or the clear-item is 'NETWORK'.

Description

The clear-statement clears a certain channel or machine feature. If the channel is specified, then a control Z is sent down the channel. Otherwise, the action taken depends upon the clear-item, as follows:

- NETWORK : the network buffer is cleared. A channel number must be specified.
- FKEYS : the function keys on channel 105 are reset to BASIC's default strings.
- FONT : resets the character font to its initial characters. Channel 102 must be open.
- SCREEN : sends a control Z to the editor (channel 0) and graphics page (101) if open.
- TEXT : sends a control Z to the text page (channel 2).
- SOUND : sends a control Z to the sound channel (103) to flush all sound queues.
- ENVELOPE: sends a control X to the sound channel (103) to flush all envelope storage.
- QUEUE : sends an Esc Z numeric-expression to the sound channel (103) to clear the sound queue specified by the numeric-expression, which must be in the range 0 to 3.

Examples

```
CLEAR £10:
CLEAR SCREEN
CLEAR QUEUE 2
```

Associated keywords

None.

CLOSE

Syntax

close-statement = 'CLOSE' channel-number

Description

The close-statement closes the EXOS channel specified by the channel-number.

Normally, the ':' is given after a channel, but the end-of-line option is useful when nothing else is to follow on the line.

Examples

```
CLOSE £3
```

Associated keywords

OPEN. See open-statement.

CODE

Syntax

```
code-statement          = 'CODE' numeric-variable? '='
                          string-expression
```

Description

The code-statement puts each byte in the string-expression into the next free consecutive locations of memory previously allocated by an allocate-statement. The numeric-variable, if given, has assigned to it the address of the first byte of the memory area. The string-expression may have no length, in which case the number of bytes free does not change, but the numeric-variable still points to the next free byte.

Examples

```
CODE END=HEX$(C1, D1, E1, C9)
```

Associated keywords

ALLOCATE, HEX$, WORD$. See allocate-statement, hex-call, word-call.

CONTINUE

Syntax

```
continue-line          = 'CONTINUE'
```

Description

The continue-line has two different actions, the action depending upon whether it is used in 'immediate mode' or in a program.

When used in immediate mode, the continue-line allows a program whose execution has previously been stopped to be continued again. The program must have been stopped as a result of a stop-statement or a press of the STOP key. The program cannot be continued if an error has occured or the program has been edited since being stopped.

When used in a program, the continue-line is used as an exit from an exception handler. Execution continues at the line after the line that caused the exception.

Examples

```
CONTINUE

2040 IF EXTYPE = 100 THEN
2050   CONTINUE
2060 END IF
```

Associated keywords

RETRY, HANDLER, EXIT HANDLER, END HANDLER, STOP. See retry-line, handler-line, exit-handler-statement, end-handler-statement, stop-statement.

COPY

Syntax

```
copy-statement         = 'COPY' source-channel?
                           destination-channel?
```

Description

The copy-statement reads characters from the source-channel and writes them to the destination-channel. The copy operation will cease when one of the following conditions is met:

- the STOP key is pressed
- an error occurs from either channel
- the end of file is reached on the source-channel

The source-channel defaults to channel 0 (EDITOR:) and the destination-channel defaults to channel 104 (PRINTER:).

Examples

```
COPY FROM £10 TO £20
```

Associated keywords

CAPTURE, REDIRECT. See capture-statement, redirect-statement.

DATA

Syntax

```
data-statement          = 'DATA' data-list
data-list               = datum (',' datum)*
datum                   = constant / unquoted-string
unquoted-string         = plain-string-character / (plain-
                          string-character unquoted-string-
                          character* plain-string-
                          character)
```

Note that the definition of an unquoted-string indicates that leading and trailing spaces are not significant.

Description

The data-statement definies a sequence of data which can later be read into variables. Data-statements may appear anywhere in a program, but the totality of the data-statements is considered to define one sequence of data. The first data-statement defines the first data in the data sequence and so on.

The execution of a data-statement results in execution continuing with the next line in the program, with no other effect.

Examples

```
DATA -3E-26, he said "IS-BASIC", "he said, ""IS-
     BASIC"""
```

Associated keywords

READ, RESTORE. See read-statement, restore-statement.

DATE

Syntax

```
date-statement          = 'DATE' string-expression
```

Description

The date-statement allows the internal date counter to be set. The string-expression must evaluate to a string containing 8 characters in the format specified by ANSI Standard X3.30, which was "YYYYMMDD".

Examples

```
DATE "19840521"     ! 21st May 1984
```

Associated keywords

DATE$. See date-call.

DEF

Syntax

```
def-line                = line-number def-statement tail
def-statement           = single-line-def / block-def
single-line-def         = single-line-numeric-def / single-
                          line-string-def
single-line-numeric-def = numeric-block-def '=' numeric-
                          expression
single-line-string-def  = string-block-def '=' string-
                          expression
```



```
block-def           = numeric-block-def / string-block-
                      def
numeric-block-def   = 'DEF' numeric-identifier
                      parameter-list?
string-block-def    = 'DEF' string-identifier
                      parameter-list?
parameter-list      = '(' parameter (',' parameter)*
                      ')'
parameter           = value-parameter / reference-
                      parameter
value-parameter     = identifier
reference-parameter = 'REF' identifier
```

Description

The def-statement allows a user-defined function to be defined. This can either evaluate a simple expression in the case of a singe-line-def, or can execute many program lines in the case of a block-def. In the latter case, the block of program-lines must be terminated with an end-def line.

When the function is called, the number and type of the arguments in the argument-list on the calling line must agree with the number and type of arguments on the def-line.

A value-parameter indicates that an expression will be evaluated as the argument, and assigned to the value-parameter, which then becomes an ordinary variable whose name and value are local to the function.

A reference-parameter indicates that the reference-parameter refers to the actual variable given in the argument-list, and then becomes an ordinary variable whose name only is local to the function. Thus if its value is changed by the function, then the value of the variable in the argument-list is also changed.

The execution of a single-line-def results in execution continuing with the next program line, with no other effect. The execution of a block-def results in execution continuing with the line immediately following the corresponding end-def-line, with no other effect.

The function named in a def-line can be accessed anywhere in the program, regardless of whether or not the def-line appears before or after the defined-function that references it. Functions named in a def-line are global to the whole program and can be accessed anywhere regardless of whether the def-line appeared within another function definition or not.

In the case of a block-def, the result is returned by assigning a value to the name of the function. This assignment may be performed more than once (in which case the last value assigned will be returned).

The name of the function may also be used in a definied-function reference within that function, thus allowing recursive functions to be written. The number of recursions allowed depends upon the context in which the function is called and the number of arguments passed, but typically approaches 60.

If a block-def is intended to be the the object of a call-statement, then no result need be assigned to the name of the function, thus providing procedural capabilities. The function may not then be used as a function returning a result since an exception will result.

Examples

```
DEF X(ANGLE) = SIN(ANGLE)*700

10 DEF XY(ANGLE, REF FUNC)
20   XY=FUNC(ANGLE)*700
30 END DEF
```

Associated keywords

END DEF, CALL. See end-def-line, call-statement, definied-function.

DELETE

Syntax

```
delete-command      = 'DELETE' segment-list?
segment-list        = segment-specifier (',' segment-
                      specifier)*
segment-specifier   = defined-function / handler /
                      line-range
line-range          = first-line / last-line
first-line          = line-specifier last-line?
last-line           = ('-' / 'TO') line-specifier?
line-specifier      = 'FIRST' / 'LAST' / line-number
```

A segment-specifier specifies a group of lines that are to be used.

If the segment-list is not given, then the whole program is used.

If the segment-specifier is a defined-function, then the group of lines referenced are from the def-line or handler-line to the corresponding end-def-line or end-handler-line respectively.

If the line-specifier is 'FIRST', then the line number of the first line of the program will be used. If the line-specifier is 'LAST', then the last line of the program will be used.

If the line-range does not include a first-line, then the first line of the range is assumed to be the first line of the program. Similarly, if the last-line does not include a line-specifier, then the last line of the program is assumed.

Description

The delete-command can be used to delete a section of the program Although just typing in just the line-number alone will delete that line, the delete-command is convenient for deleting more than one line.

If no segment-list is given, then the whole program will be deleted, which is equivalent to the new-command.

Examples

```
DELETE FIRST TO 100
DELETE -100
DELETE 400-
DELETE 300 TO 400
```

Associated keywords

LIST, NEW. See list-command, new-command.

DIM

Syntax

```
dim-statement          = 'DIM' dimension-list
dimension-list         = array-declaration (',' array-
                         declaration)*
array-declaration      = (numeric-array / string-array)
                         bounds
numeric-array          = numeric-identifier
string-array           = string-identifier
bounds                 = '(' bounds-range (',' bounds-
                         range)? ')'
```



```
bounds-range          = lower-bound? bound
lower-bound           = bound 'TO'
bound                 = numeric-expression
```

If no lower-bound is specified in a bounds-range, then a value of zero will be assumed. If the numeric-expression in the bound evaluates to a non-integral value, then it will be truncated to the nearest integer towards zero.

Description

The dim-statement declares the named arrays to be either one- or two-dimensional, depending upon whether one or two bounds-ranges are specified. The maximum subscript value is specified optionally with the minimum subscript value the array will have.

The maximum length of each element of a string array declared with a dim-statement is 132 characters. This may be changed by using a string-statement instead of a dim-statement.

Examples

```
DIM A(2 TO 10), FRED$(500)
```

Associated keywords

NUMERIC, STRING. See numeric-statement, string-statement.

DISPLAY

Syntax

```
display-statement       = 'DISPLAY' (text-display-specifier
                          / graphics-display-specifier /
                          general-display-specifier)
text-display-specifier  = 'TEXT'
graphics-display-specifier = 'GRAPHICS'
general-display-specifier= (channel ':')? display-argument*
display-argument        = ('AT' / 'TO' / 'FROM') numeric-
                          expression
```

Description

The display-statement causes a video page to be displayed.

If the display-statement has a text-display-specifier, then 24 lines of text page and editor buffer are displayed. If the graphics channel is open, then it will not be closed unless necessary. If the currently open text page is to small, then it will be closed and re-opened with the correct size.

If the display-statement has a graphics-display-specifier, then 20 lines of graphics and 4 lines of text and editor buffer are displayed. If a larger text page is open, then this will not be changed unless necessary. If a graphics page is not open, then one will be opened.

If the display-statement has a general-display-specifier, then the channel will be displayed. The default channel displayed is the graphics channel.

The last 'AT', 'TO' and 'FROM' values given in a display-argument determine which part of the video page is displayed, and where on the screen it is displayed. The default values are 1, 24 and 1 respectively. 'AT' determines where on the screen the portion of video page is to be displayed. 'FROM' specifies the first line of the video page to display, and 'TO' specifies the last line. If the 'TO' value is zero, then the border colour will be displayed for the number of lines specified by the 'TO' part.

Examples

```
DISPLAY £2:AT 10 FROM 10 TO 24.
```

Associated keywords

GRAPHICS, TEXT. See graphics-statement, text-statement.

DO

Syntax

```
do-line                 = line-number do-statement tail
do-statement            = 'DO' exit-condition?
exit-condition          = ('WHILE' / 'UNTIL') numeric-
                          expression
```

An exit-condition is said to be 'true' if the numeric-expression following the 'WHILE' is zero, or if the numeric-expression following the 'UNTIL' is non-zero.

Description

The do-statement introduces a DO...LOOP, which is a loop repeated until or while a condition is true.

When a do-line is executed, the exit-condition (if given) is evaluated, and if false the next program line is executed. If true, then execution continues with the line immediately following the associated loop-line. If no exit-condition is given, then a value of always true will be assumed.

Examples

```
10 DO UNTIL X>10
```

Associated keywords

LOOP. See loop-line.

EDIT

Syntax

edit-command = 'EDIT' program-designator

Description

The chain-command makes the program specified by the program-designator to become the current program, and the subject of subsequent program editing and execution commands. The program specified by the program-designator must currently reside in memory.

Examples

```
EDIT 10
EDIT "MY_PROG"
```

Associated keywords:

CHAIN, PROGRAM. See chain-statement, program-line.

ELSE

Syntax

else-line = line-number 'ELSE' tail

Description

The else-line is used in conjunction with the if-line and specifies a block of lines to be executed if the condition in the associated if-line is false. The block is terminated by an else-if-line or end-if-line.

Examples

```
ELSE
```

Associated keywords

IF, ELSE IF, END IF. See if-line, else-if-line, end-if-line.

ELSE IF

Syntax

else-if-line = line-number 'ELSE IF' numeric-expression tail

Description

The else-if-line allows a further conditions to be tested after a condition on an if-line has become false. Any number of else-if-lines may appear between the if-line and the associated end-if-line. The same effect as an else-if-line may be achieved by nesting if-lines, but an else-if-line is generally more compact and readable. Using nested if-lines makes a program listing rather unwieldly, particularly when lines are indented to show the program structure.

If the numeric-expression evaluates to non-zero, then execution continues with the next line in the program. If it evaluates to zero, then control passes to the next if-statement terminator. This may be another else-if-line, and else-line or an end-if-line.

Examples

```
ELSE IF A<10
```

Associated keywords

ELSE, IF, END IF. See else-line, if-line, end-if-line.

END

Syntax

```
end-line                    = line-number 'END' tail
```

Description

Terminates execution of the program, and returns to immediate mode. The program cannot be restarted the a continue-line (use the stop-statement to allow this).

Examples

```
10 END
```

Associated keywords

STOP. See stop-statement.

END DEF

Syntax

```
end-def-line                = line-number 'END DEF' tail
```

Description

The end-def-line terminates a function definition, and must be matched with an associated def-line. An esception is caused if an end-def-line is executed if the function was invoked from an expression and no value has been assigned to the function.

When an end-def-line is executed, control is passed back to the line or expression that invoked the function. All variables declared or used for the first time in the function are thrown away.

Examples

END DEF

Associated keywords

DEF. See def-line.

HANDLER

Syntax

end-handler-line = line-number 'END HANDLER' tail

Description

The end-handler-line is used to terminate an exception handler, and must always be matched by an associated handler-line.

When an exception handler is executed, the end-handler-line is not necessarily executed, since it is only one of four ways of exiting the exception handler. If it is executed, then execution continues at the line immediately following the end-when-line that terminates the when block in which the exception occured. All local variables created during execution of the handler are thrown away, together with those defined since the when-line associated with the end-when-line was executed. Also all gosub-statements, do-lines, for-lines, and function calls made active since execution of the when-line are abandoned. The effect is thus as though the when block was never executed.

Examples

END HANDLER

Associated keywords

HANDLER, WHEN, END WHEN, EXIT HANDLER, CONTINUE, RETRY. See handler-line, when-line, end-when-line, exit-handler-statement, continue-line, retry-line.

END IF

Syntax

end-if-line = line-number 'END IF' tail

Description

The end-if-line marks the end of a block of lines following an if-line, else-line or else-if-line, and must have an associated if-line. If executed, execution continues with the next program line with no other effect.

Examples

```
END IF
```

Associated keywords

ELSE, ELSE IF, IF. See else-line, else-if-line, if-line.

END SELECT

Syntax

end-select-line = line-number 'END SELECT' tail

Description

The end-select-line marks the end of a block of lines following a select-line or case-line, and must be matched by an associated select-line. When executed, an exception occurs if no case-line has been selected.

Examples

```
END SELECT
```

Associated keywords

SELECT, CASE. See select-line, case-line.

END WHEN

Syntax

```
end-when-line          = line-number 'END WHEN' tail
```

Description

The end-when-line marks the end of a block of program lines following a when-line, and must be matched by an associated when-line. If executed, execution passes to the next line with no further effect.

The line after an end-when-line is where execution continues if an end-handler-line is executed in an exception handler.

Examples

```
END WHEN
```

Associated keywords

WHEN, END HANDLER. See when-line, end-handler-line.

ENVELOPE

Syntax

```
envelope-statement     = 'ENVELOPE' (channel ':')?
                         'NUMBER' numeric-expression
                         envelope release?
envelope               = phase*
phase                  = ';' pitch-change ',' left-volume-
                         change ',' right-volume-change
                         ',' envelope-duration
pitch-change           = numeric-expression
left-volume-change     = numeric-expression
right-volume-change    = numeric-expression
envelope-duration      = numeric-expression
release                = ';RELEASE' envelope
```

Description

The envelope-statement defines an envelope to be later used with the sound-statement.

The numeric-expression following the 'NUMBER' must be in the range 0 to 254, and allows the envelope being defined to be referenced in a subsequent sound-statement.

The pitch-change specifies the change in pitch over the period of the
phase in which it appears in semitones. Fractional semitones are allowed.

The left-volume-change and the right-volume-change specify the change in volume for the left sound channel and the right sound channel respectively over the period of the phase in which they appear. They must be in the range -63 to 63 to have an effect, and represent a proportion of the overall maximum volume specified in a subsequent sound-statement. Any values outside the above range will be truncated to maximum of 63 or minimum or -63.

The envelope-duation specifies the length of the phase in which it appears in 1/50ths of a second.

The optional release specified an envelope to be used when the main envelope terminates and when the duration specified in the sound-statement has expired and there are no other sounds following in the queue.

Examples

```
ENVELOPE NUMBER 1; 10,4,5,200; 1000,40,40,503; RELEASE;
         -10,-4,-5,-200
```

Associated keywords

SOUND. See sound-statement.

EXIT DEF

Syntax

exit-def-statement = 'EXIT DEF'

Description

The exit-def-statement is used to prematurely exit the mode recently invoked defined-function. An exception occurs if the function was invoked as a result of evaluating an expression (ie. if not from a call-statement) and no value has been assigned to the function name.

The exit-def-statement can be used within blocks introduced by for-lines and do-lines, if-lines, and select-lines.

Examples

```
EXIT DEF
IF A=B THEN EXIT DEF
```

Associated keywords

DEF, END DEF. See def-line, end-def-line.

EXIT DO

Syntax

exit-do-statement = 'EXIT DO'

Description

The exit-do-statement is used anywhere in a block of program lines introduced by a do-line, and is used to prematurely exit the loop, which must be the inner-most loop.

Examples

```
EXIT DO
IF A=B THEN EXIT DO
```

Associated keywords

DO, LOOP. See do-line, loop-line.

EXIT FOR

Syntax

exit-for-statement = 'EXIT FOR'

Description

The exit-for-statement is used anywhere in a block of program lines introduced by a for-line, and is used to prematurely exit the loop, which must be the inner-most loop.

Examples

```
EXIT FOR
IF A=B THEN EXIT FOR
```

Associated keywords

FOR, NEXT. See for-line, next-line.

EXIT HANDLER

Syntax

exit-handler-statement = 'EXIT HANDLER'

Description

The exit-handler-statement is used in an exception handler when that exception handler is not interested in or cannot handler the exception. The exit-handler-statement then passes the exception to the next outer exception handler (or the system error handler if none) ie. the effect is as though the exception handler (and the associated when-line) with the exit-handler-statement in did not exist.

The exit-handler-statement can be used within blocks introduced by for-lines and do-lines, if-lines, and select-lines.

Examples

```
EXIT HANDLER
IF A=B THEN EXIT HANDLER
```

Associated keywords

END HANDLER, CONTINUE, RETRY. See end-handler-line, continue-line, retry-line.

EXT

Syntax

```
ext-statement          = 'EXT' string-expression
```

Description

The ext-statement is used to pass the string-expression around all the ROMs in the system via EXOS to perform some 'service' or to start up a new applications program to replace BASIC.

In 'immediate mode' a line beginning with a colon ':' causes the remainder of the line to be passed to EXOS in a similar manner.

Examples

```
EXT "DIR"
:HELP
```

Associated keywords

None.

FETCH

Syntax

```
fetch-statement        = 'FETCH' channel (':' record)?
record                 = numeric-expression
```

Description

The fetch-statement provides a way of randomly reading a record from a file which was created using the write-statement and record-statement. If the record is not specified, then the default of the last record read or written + 1 is used.

Examples

```
FETCH £10
FETCH £10: 47
```

Associated keywords

RECORD, WRITE. See record-statement, write-statement.

FLUSH

Syntax

```
flush-statement          = 'FLUSH' channel ':'?
```

Description

The flush statement causes a buffer on the specified channel to be written out. This is usually used to cause buffered data to be sent down the network.

Examples

```
FLUSH £10
```

Associated keywords

CLEAR. See clear-statement.

FOR

Syntax

```
for-line                 = line-number for-statement tail
for-statement            = 'FOR' control-variable '='
                           initial-value 'TO' limit ('STEP'
                           increment)?
control-variable         = simple-numeric-variable
initial-value            = numeric-expression
limit                    = numeric-expression
increment                = numeric-expression
```


Description

The for-line introduces a FOR...NEXT loop, which is a block of program lines that are executed a specified number of times with the value of the control-variable incremented on each execution of the loop. The for-line must be matched by an associated next-line.

When a for-line is executed, the initial-value is assigned to the control variable. A test to see if the value of the control-variable has exceeded the limit is then performed as for the next-line, and if the test is not true execution continues with the next program line. If the test is true, then execution continues with the line immediately following the associated next-line.

If the 'STEP' and increment are omitted, then the increment defaults to one.

Examples

```
FOR A=1 TO B STEP 2*C
```

Associated keywords

NEXT. See next-line.

GET

Syntax

```
get-statement = 'GET' (channel ':') string-
                variable
```

Description

The get-statement allows a character to be read from the channel (default 105 (KEYBOARD:) if not given) with an immediate return if no character is ready.

The character read is assigned to the string-variable. If no character was ready to be read then the string assigned to the string-variable has zero length.

Examples

```
GET A$
GET £10:CHAR$
```

Associated keywords

INKEY$. See inkey-call/

GOSUB

Syntax

gosub-statement = 'GOSUB' line-number

Description

The gosub-statement when executed causes execution to be continued at the line-number specified. When a return-statement is subsequently executed, execution continues with the program-line immediately following the one after the gosub-statement.

Care should be taken when mixing gosub-statements with definied-function invocations.

Examples

```
GOSUB 10
```

Associated keywords

GOTO, RETURN. See goto-statement, return-statement.

GOTO

Syntax

goto-statement = 'GOTO' line-number

Description

The goto-statement when executed causes execution to continue at the specified line-number.

Care should be excersised when using goto with any blocks (FOR...NEXT, DO...LOOP, DEF...END DEF, IF...THEN etc.)

Examples

```
GOTO 10
```

Associated keywords

GOSUB. See gosub-statement.

GRAPHICS

Syntax

```
graphics-statement      = 'GRAPHICS' ('ATTRIBUTE' /
                          (resolution? colour-mode?) )?
resolution              = 'HIRES' / 'LORES'
colour-mode             = numeric-expression
```

The colour-mode must evaluate to a value of 2, 4, 16 or 256.

Description

The graphics-statement sets up and displays 4 lines of text and 20 lines of graphics, which is either an attribute graphics page or the specified resolution and colour-mode.

If 'ATTRIBUTE' is not specified and no resolution is given, then it defaults to the last resolution used, or 'HIRES' initially. If 'ATTRIBUTE' is not specified and no colour-mode is given, then it defaults to the last colour-mode used, or 4 initially.

Examples

```
GRAPHICS
GRAPHICS HIRES 2
GRAPHICS LORES 256
GRAPHICS ATTRIBUTE
```

Associated keywords

DISPLAY, TEXT. See display-statement, text-statement.

HANDLER

Syntax

```
handler-line        = 'HANDLER' handler
handler             = numeric-identifier
```

Description

The handler-line introduces a block of program lines that are used as an exception handler. It must be matched by an associated end-handler-line.

After a when-line that references the handler-line (via the numeric-identifier) has been executed, any exceptions that occur cause the line immediately following the handler-line, and subsequent lines, to be executed.

The handler may be exited in one of four ways:

- a continue-line
- an exit-handler-statement
- a retry-line
- an end-handler-line

Examples

```
HANDLER IO_ERROR
```

Associated keywords

END HANDLER, EXIT HANDLER, RETRY, CONTINUE, WHEN. See end-handler-line, exit-handler-statement, retry-line, continue-line, when-line.

IF

Syntax

```
if-line             = line-number if-then-statement
                      tail
if-statement        = if-then-statement (statements /
                      line-number)
```


if-then-statement = 'IF' numeric-expression 'THEN'

The numeric-expression is said to be 'false' if it evaluates to a value of zero; otherwise it is said to be true.

Descrition

The if-statement provides the conditional execution of one or more statements. The if-line allows the conditional execution of zero or more program-lines as a block, with the option of an else-line and any numer of else-if-lines, and must be matched by an associated end-if-line.

When an if-statement is executed, the next line is executed if the numeric-expression in the if-then-statement evaluates to false. Otherwise, if a line-number is given, execution continues at the specified line-number, or if statements are specified then execution continues with the first statement in the statements. This could be another if-statement.

When an if-line is executed, the next line is executed if the numeric-expression evaluates to true. If it is false, then execution continues with the next line that is either an else-if-line, an else-line or an end-if-line.

Examples

```
IF A=B THEN 100
IF A=B THEN GRAPHICS: GOTO 100

IF A=B THEN
  PRINT "A=B"
ELSE IF C=D THEN
  PRINT "C=D"
ELSE
  PRINT "Sorry"
END IF
```

Associated keywords

ELSE, ELSE IF, END IF. See else-line, else-if-line, end-if-line.

IMAGE

Syntax

```
image-line          = line-number 'IMAGE' ':' format-
                      string end-of-line
format-string       = literal-string (format-item
                      literal-string)*
literal-string      = literal-item*
literal-item        = letter / digit / ''' / ':' /
                      '=' / '!' / '(' / ')' / ';' /
                      '/' / ' ' / '_'
format-item         = (justifier? floating-character*
                      (i-format-item / f-format-item /
                      e-format-item)) / justifier
justifier           = '<' / '>'
floating-characeter = sign / '$'
i-format-item       = digit-place digit-place* (','
                      digit-place digit-place*)*
digit-place         = '*' / '£' / '%'
f-format-item       = ('.' '£' '£'*) / (i-format-item
                      '.' '£'*)
e-format-item       = (i-format-item / f-format-item)
                      '^' '^' '^' '^'*
```

Any leading spaces immediately after the colon in the image-line are considered to be part of the format-string. The format-string in an image-line ends with the end-of-line ie. trailing spaces are considered to be part of the format-string.

All digit-places in an i-format-item are significant, but if one or more digit-places is '*' or '%', then the last '*' or '%' determines the type that the digit-places are considered to be.

Description

Format-string allows the formatted printing of numbers and strings. The maximum length of the format-string is 127 characters. The effect of the characters appearing in the format-string have the following effect:

Numeric:

< £ > - Causes a digit, leading space or trailing zero to be printed.
* - Causes a digit or leading character to be printed. Sets the leading character to '*'.
% - Causes a digit or leading character to be printed. Sets the leading character to '0'.
. - Ends the printing of the integer part of a number and prints a '.'.
- - Prints a floating '-' in front of the number if it is negative, otherwise prints a space.
+ - Prints a floating '+' or '-' sign in front of the number, depending upon whether it is positive or negative repectively.
$ - Prints a floating '$' in front of the number.
^ - Prints a character from the exponent. Note that there must be at least four.
, - Prints a comma in the number.

String:

£ % * - Prints a character from the string.
< > - Prints a character from the string, and causes the string to be left- or right-justified respectively in the field defined by £, % or *. If not given, then the string is centered (to the nearest character position) within the string. Begins a new format-item.

All other characters in the format-string are printed as literals, and separate format-items.

If there are insufficient characters in the format-item to print the item, then an exception is caused.

Examples

```
IMAGE :£££.£££^^^^
```

See print-statement.

Associated keywords

PRINT. See print-statement.

INFO

Syntax

info-command = 'INFO'

Description

The info-command prints on the screen information describing the current memory usage. The format of the printout is:

<number of bytes in system>
<number of bytes not working>
<number of bytes unused in system>

<program number> <bytes used> <first line>
<program number> <bytes used> <first line>
.
.
.
.

The number of bytes in the system is the number of bytes that are intended to work, ie. (number of bytes working) + (number of bytes not working).

The line containing the number of bytes not working is only printed if this number is non-zero. The number is in multiples of 16K.

The number of bytes unused is the number of bytes that are not used by EXOS, BASIC or the user's program. Note that this is does not imply that the current program could grow this big; the FREE function should be used to obtain this number.

The rest of the printout gives a summary of the programs in memory. Each program number that has a program assigned to it is printed, followed immediately by a '*' character if it is the current program. The number of bytes occupied by the program is then printed, followed by the first line of the program. Since this will usually be a program-name-line or a comment, an 'index' of all the programs in memory is printed.

Program zero, being a special case, will always appear on the above printout. If no program is assigned to it, zero is printed for the number of bytes used, and no program line is printed.

Note that no information about the programs in memory is printed if there are no programs at all.

Examples

INFO

Associated keywords

FREE. See free-call.

INPUT

Syntax

```
input-statement         = 'INPUT' input-control? variable-
                          list
input-control           = input-conrol-item (',' input-
                          control-item)* ':'
input-control-item      = channel / prompt-specifier /
                          missing-recovery / cursor-at
prompt-specifier        = 'PROMPT' string-expression
missing-recovery        = 'IF MISSING' recovery-action
recovery-action         = exit-def-statement / exit-do-
                          statement / exit-for-statement /
                          line-number
cursor-at               = 'AT' numeric-expression ','
                          numeric-expression
variable-list           = variable (',' variable')
input-prompt            = '? '
input-reply             = data-list end-of-line
```

The cursor-at positions the cursor at the co-ordinates specified by the numeric-expressions in the cursor-at. The first numeric-expression specifies the row, and the second numeric-expression specifies the column. The first row or column is numbered 1. A row or column of zero indicates that the current cursor row or column is to be used.

Description

The input-statement is used to input data external to the program (ie. from a channel) into numeric-variables or string-variables.

If no channel is given in the input-control, then the default channel of 0 (EDITOR:) is used.

If a prompt-specifier is not given, then the default input-prompt is used, Otherwise, the string-expression in the prompt-specifier is evaluated and used as a prompt.

After receiving the input-reply, each datum is evaluated and assigned to the corresponding variable in the variable-list. If the input-reply contains no datum, or if the end of file is reached from the channel from which the input-reply is read, then an exception occurs unless a missing-recovery is specified, in which case the specified action is taken. If the missing-recovery is a line-number, then an implied goto-statement is assumed. If the input-reply does contain at least one datum, but the number of data is less than the number of variables in the variable-list, then the prompt is re-printed and a new input-reply is received. Note that this does not re-evaluate the string-expression in the prompt-specifier.

Each type of datum in the input-reply must be the same as the type of the corresponding variable in the variable-list. Note that an unquoted-string mey be either type.

Examples

```
INPUT A
INPUT £3:A$
INPUT £10, AT 1,1, PROMPT "Name :", IF MISSING 100: N$
```

Associated keywords

LINE INPUT, READ, DATA. See line-input-statement, read-statement, data-line.

LET

Syntax

```
let-statement = 'LET'? variable-list '='
                expression
```

Description

The let-statement assigns a value to one or more variables.

The type of all the variables in the variable-list must be the same. The type of the expression must be the same as the type of the variables in the variable-list.

All the subscripts in the variable-list are evaluated from left to right before the expression is evaluated. The value of the expression is then assigned to the variables.

Examples

```
LET A=1
A=1
LET A, B, FRED, A(10) = FRED+1
A$, B$, C$ = ""
```

Associated keywords

None.

LINE INPUT

Syntax

```
line-input-statement    = 'LINE INPUT' input-control?
                          string-variable
line-input-reply        = character* end-of-line
```

Description

The line-input-statement assigns the whole line-input-reply to the specified string variable. This includes all ',', ' ' and '"' characters.

The input-control is evaluated and used as in the input-statement. In the case of the line-input-statement, the missing-recovery action is only taken if the end of file is reached from the channel from which the line-input-reply is read. Thus it is possible to input a null string.

Examples

```
LINE INPUT NAME$
```

Associated keywords

INPUT. See input-statement.

LIST

Syntax

```
list-command = 'LIST' channel / ( (channel ':')?
               segment-list? )
```

Description

The list-command is used to list all or part of the current program to a channel.

If channel is not specified, then the default channel of 0 (EDITOR:) is used.

By specifying a channel, a file containing the ASCII representation of the program can be obtained. This is in contrast the save-command, which outputs the program in an internal representation.

Examples

```
LIST
LIST -100
LIST FRED
LIST £10
LIST £10:100 TO LAST
```

Associated keywords

DELETE, SAVE. See delete-command, save-command.

LLIST

Syntax

```
llist-command = 'LLIST' channel / ((channel ':')?
                segment-list?)
```

Description

The llist-command is used to list all or part of the program to the printer.

The llist-command lists the program in the same way as the list-command, except that if channel is not given then channel 105 (PRINTER:) is used.

Examples

```
LLIST
LLIST FIRST TO 400
```

Associated keywords

LIST. See list-command.

LOAD

Syntax

```
load-command        = 'LOAD' load-channel? file-name
load-channel        = channel ':'
file-name           = string-expression
```

Description

The load-command is used to load BASIC programs, BASIC extensions, system extensions and data from channels.

If loading BASIC programs, the load-command is able to load programs saved in BASICs internal program representation with the save-command, or in ASCII saved with the list-command. In the latter case, the ASCII should contain no 'immediate-mode' commands.

If the program was saved with an all-option, then all programs in memory will be deleted before loading the program. If the program was not saved with an all-option, then only the current program will be deleted before loading the new one.

When loading BASIC extensions, some RAM may be taken away from the RAM usable by program 0.

When loading System Extensions, some extra RAM may be taken by the system.

If a load-channel is specified, then data can be loaded to the specified channel. This is so that the contents of a previously saved editor buffer or video page can be loaded. In the case of a video page, the channel being loaded to must be the same size, mode and colour mode as the original page.

Examples

```
LOAD
LOAD "TAPE:MY_PROG"
LOAD £101: "PICTURE"
```

Associated keywords

MERGE, SAVE. See merge-command, save-command.

LOOK

Syntax

```
look-line          = line-number look-statement tail
look-statement     = 'LOOK' look-list? numeric-
                     variable
look-list          = look-item (',' look-item)* ':'
look-item          = beam-at / channel
beam-at            = 'AT' numeric-expression ','
                     numeric-expression
```

Description

The look-line assigns to a numeric-variable the ordinal position of the character at the current cursor position or the palette colour of the point at the current beam position depending upon whether the channel is a video text page or graphics page, repectively. If the channel is not given as a look-item, then the default channel of 101 (GRAPHICS:) is used.

If the beam-at is given, then the beam is turned off and moved to the co-ordinates specified by the numeric-expressions. The first numeric-expression is the x co-ordinate, and the second numeric-expression is the y co-ordinate. The beam-at should not be used if the channel refers to a text page.

Examples

```
LOOK A
LOOK £10, AT 500,500: A
```

Associated keywords

None.

LOOP

Syntax

```
do-line                 = line-number do-statement tail
do-statement            = 'LOOP' exit-condition?
```

Description

The loop-line ends a DO...LOOP, which is a loop repeated while or until a condition is true. It must be matched by an associated do-line.

When the loop-line is executed, the exit-condition (if given) is evaluated, and if true the next program line is executed. If true, then the associated do-line is executed.

Examples

```
LOOP
LOOP WHILE X<10
```

Associated keywords

DO. See do-line.

LPRINT

Syntax

```
lprint-statement        = 'LPRINT' print-control? print-
                          list?
print-control           = print-control-item (',' print-
                          control-item)* ':'
print-control-item      = channel / cursor-at / using-
                          specifier
using-specifier         = 'USING' (string-expression /
                          line-number)
```



```
print-list              = (print-item? print-separator)*
                          print-item?
print-item              = expression / tab-call
tab-call                = 'TAB' '(' numeric-expression ')'
print-separator         = ',' / ';'
```

Description

The lprint-statement allows data to be printed to a printer.

The print-control and print-list are evaluated as for the print-statement, except that if the channel in the print-control-item is not specified then a default of 105 (PRINTER:) will be used.

Examples

```
LPRINT
LPRINT USING HOURS$: TIME$(:2)
```

Associated keywords

PRINT. See print-statement.

MERGE

Syntax

```
merge-command           = 'MERGE' file-name?
```

Description

The merge-command allows a program not currently in memory to be combined with the current program. The program must either be in ASCII, or must have been saved without the all-option.

Each line from the program specified by the file-name is taken in turn, and inserted into the program as though it had been typed in in 'immediate mode' ie. any existing line with the same line number is first deleted, and then the new line is inserted into the program at the appropriate position.

Examples

```
MERGE
MERGE "2:SUBROUTINES"
```

Associated keywords

LOAD. See load-command.

NEW

Syntax

```
new-command                = 'NEW' all-option?
all-option                 = 'ALL'
```

Description

The new-command erases programs from memory.

If the all-option is not specified, then the current program and all its variables is deleted from memory.

If the all-option is specified, then all programs in memory are deleted, and program 0 is made the current program.

Examples

```
NEW
NEW ALL
```

Associated keywords

DELETE. See delete-command.

NEXT

Syntax

```
next-line                  = line-number next-statement tail
next-statement             = 'NEXT' simple-numeric-variable?
```


Description

The next-line ends a FOR...NEXT loop, and must be matched by an associated for-line with a control-variable the same as the simple-numeric-variable, if given.

When the next-line is executed, the increment of the associated for-line is added to the current control-variable value. The following condition is then evaluated:

(control-variable value - limit) * SGN(increment) > 0

If this condition is zero (false) then execution continues at the line immediately following the associated for-line. Otherwise, execution continues with the next program line.

Examples

```
NEXT
NEXT X
```

Associated keywords

FOR. See for-line.

NUMERIC

Syntax

```
numeric-line          = line-number numeric-statement
                        tail
numeric-statement     = 'NUMERIC' numeric-declaration
                        (',' numeric-declaration)*
numeric-declaration   = numeric-identifier bounds?
```

Description

The numeric-line is used for the declaration of numeric-variables and numeric-arrays.

Typical use of the numeric-line is to declare local simple-numeric-variables and numeric-arrays inside a defined-function. Since the scope of variables in IS-BASIC is dynamic, this ensures that the variable really will be local, independant upon the context in which the defined-function is invoked.

Examples

```
NUMERIC A, B, C(2 TO 4)
```

Associated keywords

DIM, STRING. See dim-line, string-line.

OK

Syntax

```
ok-command                = 'OK'
```

Description

The ok-command is provided so that when using BASIC with the screen editor, the cursor can be 'returned' over an 'ok' printed previously by BASIC without causing an exception. The ok-command does not do anything.

Examples

```
ok
```

Associated keywords

None.

ON

Syntax

```
on-line                   = line-number (on-goto-statement /
                            on-gosub-statement) tail
on-goto-statement         = 'ON' numeric-expression 'GOTO'
                            line-number-list
on-gosub-statement        = 'ON' numeric-expression 'GOSUB'
                            line-number-list
line-number-list          = line-number (',' line-number)*
```


Description

The on-line allows execution to conditionally continue at another selected program line. If the on-line contains an on-goto-statement, then the effect of continuing execution at the new line will be the same as if execution had continued there as a result of executing a goto-statement. If the on-line contains an on-gosub-statement, then the effect of continuing execution at the new line will be the same as if execution had continued there as a result of executing a gosub-statement. In the latter case, a subsequent return-statement will cause execution to continue at the line immediately following the original on-line.

The numeric-expression is first evaluated, and the program line at which execution will continue is selected from the line-number-list based on its value. If the numeric-expression evaluates to n, then execution will continue at the nth line number in the list. n=1 => the first line number in the line-number-list. If n exceeds the number of line-numbers in the line-number-list or is less than one, then execution continues with the next program line.

Examples

```
ON A GOTO 100,200,300
ON INK GOSUB 103, 467, 765, 654
```

Associated keywords

GOSUB, GOTO, SELECT. See gosub-statement, goto-statement, select-line.

OPEN

Syntax

```
open-statement      = 'OPEN' channel ':' open-name
                      access? record-number
open-name           = 'NAME'? file-name
access              = 'ACCESS' ('INPUT' / 'OUTPUT')
record-number       = 'RECORD' numeric=expression
```

Description

The open-statement opens an I/O channel to a device and possibly file via the operating system with the channel number specified by channel.

If the access is 'OUTPUT' then a new file is created. If the access is 'INPUT' then an existing file is opened. If no access is specified, 'INPUT' is assumed.

If the record-number is specified, then the file is opened/created for random record reading and writing. The record-number must then refer to a record structure that has previously been defined using the record-statement.

Examples

```
OPEN £10: "FRED"
OPEN £11: NAME "TAPE:MY_PROG" ACCESS OUTPUT
OPEN £10: "MY_DATA" RECORD 10
```

Associated keywords

CLOSE, RECORD. See close-statement, record-statement.

OPTION

Syntax

```
option-statement = 'OPTION' 'ANGLE' ('DEGREES' /
                   'RADIANS')
```

Description

The option-statement specifies the angle unit (degrees or radians) used in subsequent numeric-supplied-function invokation. The angle unit in effect at the time affects both the arguments to the function and the result returned.

Initially, the angle unit in effect is is radians.

Examples

```
OPTION ANGLE DEGREES
```

Associated keywords

None.

OUT

Syntax

```
out-statement          = 'OUT' port ',' numeric-expression
port                   = numeric-expression
```

Port must evaluate to a value in the range 0 to 255, and is truncated to an integer.

Description

The out-statement outputs the value of the numeric-expression to the Z80 I/O port specified by port. The numeric-expression in the out-statement must evaluate to a value in the range 0 to 255 and is truncated to an integer.

Examples

```
OUT 3,100
```

Associated keywords

IN. See in-call.

PING

Syntax

```
ping-statement         = 'PING'
```

Description

The ping-statement causes a 'ping' sound to be emitted.

Examples

```
PING
```

Associated keywords

None.

PLOT

Syntax

```
plot-statement      = 'PLOT' (channel ':')? plot-list?
plot-list           = plot-item (plot-separator plot-
                      item?)*
plot-item           = paint-item / relative-plot-item /
                      ellipse-item / beam-position
paint-item          = 'PAINT'
relative-plot-item  = ('BACK' / 'FORWARD' / 'RIGHT' /
                      'LEFT' / 'ANGLE' ) numeric-
                      expression
ellipse-item        = 'ELLIPSE' numeric-expression ','
                      numeric-expression
beam-position       = numeric-expression ',' numeric-
                      expression
plot-separator      = ',' / ';'
```

Description

The plot-statement plots dots, lines and shapes on the graphics screen specified by channel. If not given, then the default of 101 (graphics VIDEO:) is used.

Each plot-item is separated by a plot-separator. If the plot-separator is a ',', then the beam is turned off after executing the plot-item. If the plot-separator is a ';', then the beam is turned on after executing the plot-item. The beam is similarly left on or off if the plot-statement ends in a plot-separator. If it does not end in a plot-separator, then the beam is turned on and then off, thus plotting a point. If the beam is moved whilst the beam is on, then a line is drawn.

If a paint-item is specified, then the screen is filled in the current ink colour up to any boundary which is not the same colour as that at the current beam position.

If an ellipse-item is specified, then an ellipse or circle is drawn. The first numeric-expression in the ellipse-item specifies the radius on the x-axis, and the second numeric-expression specifies the radius on the y-axis.

If a beam-position is specified, then the beam is moved to the position specified by the numeric-expressions. The first numeric-expression is the x co-ordinate and the second is the y co-ordinate. If the beam is on when the beam moves, then a line is drawn.

The relative-plot-items allow lines to be drawn using 'turtle graphics'. That is, an imaginary turtle is moved around the screen, and can be rotated to face any direction. The beam is always moved with the turtle: moving the turtle with the beam on thus draws a line.

'FORWARD' and 'BACK' in a relative-plot-item move the turtle in the direction in which it is facing or the direction opposite that in which it is facing respectively the number of co-ordinates specified by the numeric-expression. 'LEFT' and 'RIGHT' rotate the turtle anit-clockwise or clockwise respectively to make it face in a different direction. The numeric-expression in the current angle unit as specified by the option-line, or radians initially. 'ANGLE' sets the turtle facing a specified absolute angle in the current angle unit. An angle of zero represents the direction of the positive y-axis, and is the initial direction of the turtle. The angle specified by the numeric-expression is measured anti-clockwise from the positive x-axis.

Examples

```
PLOT      ! plots a point
PLOT 100,100; 300,300; 150,150, PAINT
PLOT ELLIPSE 100,100,
PLOT ANGLE 0; FORWARD 50; LEFT 90; FORWARD 50;
```

Associated keywords

SET. See set-statement.

POKE

Syntax

poke-statement	= 'POKE' address, numeric-expression
address	= numeric-expression

The address must evaluate to a value in the range 0 to 65535, and is truncated to an integer.

Description

The poke-statement sets the byte at address to the value of the numeric-expression, which must evaluate to a value in the range 0 to 255, and is truncated to an integer.

Examples

```
POKE FRED, 0
```

Associated keywords

SPOKE, PEEK, SPEEK. See spoke-statement, peek-statement, speek-statement.

PRINT

Syntax

print-statement = 'PRINT' print-control? print-list?

Description

The print-statement allows data to be printed to a channel. If no channel is specified in the print-control, then the default channel of 0 (EDITOR:) is used.

A using-specifier allows data to be output in a user-specified format. The string-expression is evaluated and is used as a format-string (see image-line). If a line-number is specified in a using-specifier, then the referenced program-line must contain an image-line, and the format-string on that image-line will be used instead of the string-expression.

The print-items in the print-list are separated by one or more print-separators. If the print-separator is ',', then an ASCII TAB character (code 9) is output. If the print-separator is a ';' then nothing is output between the print-items. If the print-statement ends in a print-separator, then no carriage-return / line feed will be output, otherwise these are output at the end of all the output generated by the print-items.

If a print-item is an expression, then the value of that expression is converted to ASCII and output. If the print-item is a tab-call, then the cursor is positioned to the column specified by the numeric-expression. The first column is considiered to be column one.

Examples

```
PRINT
PRINT A; B; C ,,,D; E; F
PRINT £10, AT 10,10, USING "££££.££": PAY
PRINT NAME$; TAB(20); ADDRESS$
```



```
PRINT USING 100:A, B
PRINT USING "£££.£££":1          prints "  1.000"
PRINT USING "£££.£££":0.1        prints "   .100"
PRINT USING "£££.£££^^^":1       prints "100.000E-02"
PRINT USING "$££":1              prints " $1"
PRINT USING "+££":1              prints " +1"
PRINT USING "-££":1              prints "  1"
PRINT USING "+££":-1             prints " -1"
PRINT USING "-££":-1             prints " -1"
PRINT USING "*££££.£££":1        prints "***1.000"
PRINT USING "%£££":1             prints "0001"
PRINT USING "N=££ K=££":1,2      prints "N= 1 K= 2"
PRINT USING "££,£££,£££":1E+7    prints " 1,000,000"
PRINT USING "*££":1,2            prints "**1"
                                        "**2"
PRINT USING "<£££££££":"FRED"    prints "FRED    "
PRINT USING ">£££££££":"FRED"    prints "    FRED"
PRINT USING "£££££££":"FRED"     prints "  FRED  "
```

Associated keywords

IMAGE, LPRINT. See image-line, lprint-statement.

RANDOMIZE

Syntax

randomize-statement = 'RANDOMIZE'

Description

The randomize-statement re-seeds BASIC's pseudo-random number generator. Normally, when a program is run, the same sequence of 'random' numbers is generated. If a randomize-statement is executed near the beginning of the program, then the sequence is set to an unpredictable point, thus producing different 'random' numbers each time the program is run.

Examples

```
RANDOMIZE
```

Associated keywords

RND. See rnd-call.

READ

Syntax

```
read-statement      = 'READ' read-control? variable-
                      list
read-control        = missing-recovery ':'
```

Description

The read-statement is used to input data intenal to the program (ie. from data-statements) into numeric-variables or string-variables.

The data is read from the sequence of data as defined by data-statements. The data is assigned to the variables in the variable-list as described in the input-statement, except that the missing-recovery action will be taken if an attempt is made to read beyond the last datum in the last data-statement.

Examples

```
READ A
READ IF MISSING EXIT DO:A, B$, FRED$
```

Associated keywords

DATA, INPUT. See data-line, input-statement.

RECORD

Syntax

```
record-statement    = 'RECORD' numeric-expression
                      record-item
record-item         = ',' variable
```

Description

The record-statement defines a record format for random record reads and write to a file. The record-statement must precede the open/create operation on the file.

The numeric-expression identifies the record structure, and allows more than on open-statement to refer to the same record structure.

The variables allow each field within the record to be refered to mnemonically. Data is assigned to the fields simply by assigning data to the variables specified in the record-statement. In the case of string-variables given in the record-item, the declared maximum length of the string-variable is used to define the size of the field.

After assigning data to all the fields required, the record can be written to a file using the write-statement.

Examples

```
10 STRING NAME$*20, ADDRESS$*50
20 NUMERIC AGE
30 RECORD 3, NAME$, ADDRESS$, AGE
40 OPEN £10: "MY_DATA", ACCESS OUTPUT, RECORD 3
```

Associated keywords

FETCH, OPEN, RESET, WRITE. See fetch-statement, open-statement, reset-statement, write-statement.

REDIRECT

Syntax

```
readirect-statement      = 'REDIRECT' source-channel?
                           destination-channel?
```

Description

The redirect-statement causes any characters written to the source-channel to be sent instead to the destination-channel. The redirect operation will cease when one of the following conditions occurs:

- the STOP key is pressed
- an error occurs from the destination-channel
- a subsequent redirect-statement is performed with the same source-channel but with a destination channel of 255.

The source-channel defaults to 0 (EDITOR:) and the destination-channel defaults to 104 (PRINTER:).

Examples

```
REDIRECT
REDIRECT FROM £104 TO £10
REDIRECT FROM £104 TO £255
```

Associated keywords

CAPTURE, COPY. See capture-statement, copy-statement.

REM

Syntax

```
rem-statement          = 'REM' remark-string end-of-line
```

Description

The rem-statement indicates the end of the statements on a program-line, and introduces a comment. When a rem-statement is executed, execution continues with the next program line with no further effect.

For more flexible commenting, 'pling comments' are recommended and are introduced with '!'. They are not considered to be separate statements and so mey be used with lines and commands.

Examples

```
PRINT:PRINT:  REM new lines
```

Associated keywords

See tail.

RENUMBER

Syntax

```
renumber-command       = 'RENUMBER' renumber-item*
renumber-item          = at-number / step-size / segment-
                         specifier
```


Description

The renumber-command renumbers all line-numbers in the program, both those occuring within statements and lines and those occuring at the start of a program-line.

The segment-specifier specifies the range of program lines to renumber, and defaults to renumbering the whole program If only a segment of the program is renumbered, then references to the renumbered lines will still be changed throuout the whole program.

The at-number specifies the new line-number for the first line in the segment-specifier, and defaults to 100 unless a defined-function or handler is being renumbered, in which case it defaults to the current line number of the first line of the definied-function or handler.

The step-size specifies the increment to add to the at-number for each subsequent program line that is renumbered, and defaults to 10.

Examples

```
RENUMBER
RENUMBER MY_HANDLER STEP 5
RENUMBER 400 TO LAST AT 1000 STEP 20
```

Associated keywords

None.

RESET

Syntax

```
reset-statement          = 'RESET' channel ':' reset-item
reset-item               = 'END' / numeric-expression
```

Description

The reset-statement is used for moving the file pointer on a file. It is used for random access on the file down the specified channel when not using the record structuring facilities for BASIC. The two should not be mixed up.

If 'END' is specified in the reset-item, then the file pointer is moved to the end of the file. Otherwise, the numeric-expression is the new value of the file pointer.

Examples

```
RESET £10:END
RESET £11: R*C
```

Associated keywords

POINTER. See pointer-call.

RESTORE

Syntax

```
restore-statement        = 'RESTORE' line-number?
```

Description

The restore-statement changes the position in the data sequence defined by data-statements that the next read-statement will read data from.

The next data read will be from the next data-line on or after the specified line-number. If the line-number is not specified, then the next data read will be the first datum in the first data-statement in the program.

Examples

```
RESTORE
RESTORE 534
```

Associated keywords

DATA, READ. See data-statement, read-statement.

RETURN

Syntax

```
return-statement         = 'RETURN'
```


Description

The return-statement causes execution to continue at the program-line immediately following the last gosub-statement executed that has not been the subject of a return-statement.

Care should be exercised when mixing gosub-statements with the DO...LOOPs, FOR...NEXT loops, DEF...END DEFs etc.

Examples

```
RETURN
```

Associated keywords

GOSUB. See gosub-statement.

RUN

Syntax

run-statement = 'RUN' file-name? argument-list?

Description

The run-statement initiates execution of the current program if the file-name is not given, or the program in the file named by the file-name otherwise.

If the file-name is given, then the named program is loaded as in the load-command.

The argument-list, if given, is passed to the program to be run as in the chain-statement.

Examples

```
RUN
RUN "TAPE:MY_PROG" (100)
```

Associated keywords

CHAIN, LOAD. See chain-statement, load-command.

RETRY

Syntax

```
retry-line          = line-number 'RETRY' tail
```

Description

The retry-line is used as an exit from an exception handler. Execution continues at the start of the line that caused the exception ie. the erroneous program-line is re-executed.

Examples

```
10 RETRY
```

Associated keywords

CONTINUE, HANDLER, END HANDLER, EXIT HANDLER. See continue-line, handler-line, end-handler-line, exit-handler-statement.

SAVE

Syntax

```
save-command        = 'SAVE' save-option? file-name?
save-option         = all-option / (channel ':')
```

Description

The save-command saves a program to a file in an inernal representation of the program (ie. not ASCII).

If the all-option is given, then all programs in memory are saved to the file. Otherwise, only the current program is saved.

If the file-name is given, then the file and device specified in it are used. If no file-name is given, then BASIC looks to see if it can generate a file-name from a program-name-line. If the all-option is not given, then a program-name given on the first line of the current program may be used; otherwise a program-name given on the first line of program number 0 may be used. If the program-name is a quoted-string, then that is used as the file-name. Otherwise a string of zero length is used as the file-name.

If a channel is given as a save-option, then data from the specified channel is saved. This enables the contents of an editor buffer or a video page to be saved and later re-loaded to a similar buffer or page.

Examples

```
SAVE
SAVE ALL "TAPE2:MY_PROGS"
SAVE £101: "PICTURE"
```

Associated keywords

LIST, LOAD, MERGE, VERIFY. See list-command, load-command, merge-command, verify-command.

SELECT

Syntax

```
select-line          = line-number 'SELECT' 'CASE'?
                       expression tail
```

Description

The select-line introduces a group of blocks or program-lines which are executed according to the expression. It must be matched by an associated end-select-line.

When executed, each associated case-line is found, and the condition expression compared to the case-items in the case-list as specified for the case-line. When the expression matches a case-line, execution continues with the program-line immediately following that line. If execution then reaches another case-line, then execution continues with the line immediately following the associated end-select-line.

Examples

```
SELECT A
SELECT CASE A$(4:5)
```

Associated keywords

CASE, END SELECT. See case-line, end-select-line.

SET

Syntax

```
set-statement           = 'SET' exos-variable-set / escape-
                          sequence-set / fkey-set
exis-variable-set       = exos-variable exos-variable-value
exos-variable-value     = 'ON' / 'OFF' / numeric-expression
escape-sequence-set     = (channel ':')? escape-sequence-
                          text numeric-expression (','
                          numeric-expression)*
escape-sequence-text    = 'ATTRIBUTES' / 'PALETTE' /
                          'COLOR' / 'COLOUR' / 'PAPER' /
                          'INK' / 'CHARACTER' /
                          'CURSOR CHARACTER /
                          'CURSOR COLOR' /
                          'CURSOR COLOUR' / 'SCROLL ON' /
                          'SCROLL OFF' / 'SCROLL UP' /
                          'SCROLL DOWN' / 'BEAM ON' /
                          'BEAM OFF' / 'LINE STYLE' /
                          'LINE MODE'
fkey-set                = 'FKEY' numeric-expression string-
                          expression
```

Description

The set-statement alters a system parameter. This is done either by altering the value of an EXOS variable, sending an escape sequence to a channel, or by a special function call in the case of the fkey-set.

If an exos-variable-set is specified, then the new value written to the EXOS variable is the number-expression in the exos-variable-value. If the exos-variable-value is 'ON' or 'OFF', then a value of 0 or 255 respectively is used.

If an escape-sequence-set is specified, then an escape sequence is sent to the channel, if specified, followed by as many numeric-expressions the particular escape sequence requires. Any if the number of numeric-expressions is less than required by a particular excape sequence, then the rest of the parameters all default to 0. The escape-sequence-text correspond to the escape sequences described in the EXOS documentation.

If an fkey-set is specified, then the function key indicated by the numeric-expression is programmed with the string-expression. Function keys are numbered from one.

Examples

```
SET £10:INK 1
SET CHARACTER 1,2,4,8,16,32,64,128
SET CURSOR CHARACTER 10
```

Associated keywords

ASK, TOGGLE. See ask-statement, toggle-statement.

SOUND

Syntax

```
sound-statement     = 'SOUND' (channel ':') sound-list
sound-list          = sound-item ((',' / ';') sound-
                      item)*
sound-item          = 'INTERRUPT' / ( ('PITCH' /
                      'DURATION' / 'LEFT' / 'RIGHT' /
                      'SOURCE' / 'STYLE' / 'ENVELOPE' /
                      'SYNC') numeric-expression)
```

Description

The sound statement send the appropriate escape sequence to the channel if specified to produce a sound using an envlope previously defined using the envelope-statement. If channel is not given, then the default of 103 (SOUND:).

'PITCH' specifies the overall pitch, and must be in the range 0 to 127. In the range 0 to 83, and increase of one represents an increase in pitch of one semitone. Fractional pitch values can be given. The default is 37 (middle C).

'DURATION' specifies the duration of the non-release phase of the envelope on 1/50ths of a second. The default is 1 second.

'LEFT' and 'RIGHT' specify the overall volume for the left and right channels respectively. Both must be in the range 0 (silence) to 255, and are truncated to integers. The default is 63.

'SOURCE' specifies the tone generator used, and must be in the range 0 to 3. 3 indicates the noise channel. The default is 0.

'STYLE' specifys various effects that can be applied. It must be in the range 0 to 255 and is truncated to an integer. The default is 0.

'ENVELOPE' specifies the evelope to be used. Values 0 to 254 refer to envelopes which must have been previously defined using the envelope-statement. Envelope number 255 refers to a 'built-in' envelope, and is the default.

'SYNC' allows the start of the sound to synchronised with sounds on other channels. The default is 0.

'INTERRUPT' if given causes the sound being defined to overide any sound on the same channel that may already be going. Otherwise, the new sound will be added to the queue.

Examples

```
SOUND
SOUND £10:DURATION 10, PITCH 1000
```

Associated keywords

ENVELOPE. See envelope-statement.

SPOKE

Syntax

spoke-statement = 'SPOKE' segment ',' address ',' numeric-expression

segment = numeric-expression

Segment must be evaluate to a value in the range 0 to 255, and is truncated to an integer.

Description

The spoke-statement sets the byte at address (ANDed down to a value in the range 0 to 16383) within the segment to the value of the numeric-expression, which must be in the range 0 to 255 and is truncated to an integer.

Examples

```
SPOKE 255, POINTER, 0
```

Associated keywords

POKE, SPEEK, PEEK. See poke-statement, speek-statement, peek-statement.

START

Syntax

start-command = 'START'

Description

The start-command runs the current program if there is one program in memory. Otherwise, a program is loaded with a file-name of a string with zero length as specified in the load-command.

Examples

```
START
```

Associated keywords

LOAD, RUN. See load-command, run-statement.

STOP

Syntax

stop-statement = 'STOP'

Description

The stop-statement halts execution of the program, printing the line at which the program was stopped. It may be subsequently continued again with the continue-statement.

Examples

```
STOP
```

Associated keywords

CONTINUE. See continue-line.

STRING

Syntax

```
string-line          = line-number string-statement tail
string-statement     = 'STRING' length-max? string-
                       declaration (',' string-
                       declaration)*
length-max           = '*' integer
string-declaration   = string-identifier bounds? length-
                       max?
```

The integer in length-max must be in the range 0 to 254.

Description

The string-line is used to declare simple-string-variables and string-arrays. It can also be used to declare the maximum length of a string.

Typical use of the string-line is to declare local simple-string-variables and string-arrays inside a defined-function. Since the scope of variables in IS-BASIC is dynamic, this ensures that the variable really will be local, independant upon the context in which the defined-function is invoked.

The length-max in the string-declaration, if given, specifies the maximum number of characters that the string can contain. If it is not given, then the length-max specified in the string-statement is used. If this is not specified, then the default of 132 characters is used.

Examples

```
STRING A$, B$
STRING *10 C$, D$(12,12)*100
```

Associated keywords

DIM, NUMERIC. See dim-line, numeric-line.

TEXT

Syntax

text-statement = 'TEXT' numeric-expression?

Description

The text-statement sets up and displays 24 lines of text. If the numeric-expression is given, then it must evaluate to a value of 40 or 80, in which case 40 or 80 columns respectively are set up. If it is not given, then the last value used, or 40 initially, is assumed.

Examples

```
TEXT
TEXT 80
```

Associated keywords

DISPLAY, GRAPHICS. See display-statement, graphics-statement.

TIME

Syntax

time-statement = 'TIME' string-expression

Description

The time-statement allows the internal time counter to be set. The string-expression must evaluate to a string containing 8 characters in the format specified by ANSI Standard X3.43 which is "HH:MM:SS".

Examples

```
TIME "21:06:45"      ! 9 hrs, 6 mins, 45 secs pm.
```

Associated keywords

TIME$. See time-call.

TOGGLE

Syntax

```
toggle-statement        = 'TOGGLE' exos-variable
```

Description

The toggle-statement one's complements the specified exos-variable. For many EXOS variables, this has the effect of toggling the state of some system feature.

Examples

```
TOGGLE KEY CLICK
```

Associated keywords

ASK, SET. See ask-statement, set-statement.

TRACE

Syntax

```
trace-statement         = 'TRACE' ('ON' / 'OFF') trace-
                          channel?
trace-channel           = 'TO' channel
trace-output            = '<' line-number '>'
```


Description

The trace-statement allows the execution path of the program to be seen.

When 'ON' is given, then every time a new line is executed, the line-number is output in the format of the trace-output to the channel specified in the trace-channel. If the trace-channel is not given, the the default of 0 (EDITOR:) is used.

This feature is turned off when 'OFF' is given.

Examples

```
TRACE ON TO £10
TRACE OFF
```

Associated keywords

None.

TYPE

Syntax

type-statement = 'TYPE'

Description

The type-statement exits BASIC, and enters the Enterprise WP (word processor) program. If used in 'immediate mode' then an 'Are you sure' type prompt is printed, since exiting BASIC erases all programs in memory.

Examples

```
TYPE
```

Associated keywords

EXT. See ext-statement.

VERIFY

Syntax

```
verify-command      = 'VERIFY' verify-channel file-
                      name?
verify-channel      = channel ':'
```

Description

The verify-command compares the file specified by the file-name with a program or channel in memory, and causes an exception if they are not the same.

The file may be either an application saved with the all-option in the save-command, in which case all programs are verified, a singe program saved without the all-option in which case only the current program is verified, or a channel (when channel is given) in which case data on that channel is verified. The latter form is used for verifying a file containing previously saved editor buffer or video page contents.

If no file-name is given, then a string with zero length is used as the file name.

Examples

```
VERIFY
VERIFY "MY_PROG"
VERIFY £101: "PICTURE"
```

Associated keywords

LOAD, SAVE. See load-command, save-command.

WAIT

Syntax

```
wait-statement      = 'WAIT' 'DELAY'? numeric-
                      expression
```


Description

The wait-statement pauses for the number of seconds indicated by the numeric-expression.

Examples

```
WAIT DELAY 5
```

Associated keywords

None.

WHEN

Syntax

```
when-line                = line-number when-statement tail
when-statement           = 'WHEN' 'EXCEPTION' 'USE' handler
```

Description

The when-line marks the start of a block of program lines terminated by an associated end-when-line.

When a when-line is executed, execution continues with the next program line with no further effect. If an exception occurs on a program line between the when-line and the associated end-when-line, however, then execution will continue with the named handler, which may then handle the exception. This may cause execution to continue again with the main program-lines (via the continue-statement or retry-line) or may cause execution to continue elsewhere (via the exit-handler-statement or end-handler-line).

Examples

```
10 WHEN EXCEPTION USE MY_HANDLER
```

Associated keywords

HANDLER, END HANDLER, END WHEN. See handler-line, end-handler-line, end-when-line, continue-statement, retry-line, exit-handler-line.

WRITE

Syntax

write-statement = 'WRITE' channel (':' record)?

Description

The write-statement allows the data in the current record for the specified channel to be written to the channel as the specified record.

If record is omitted then the default of the last record read/written is used.

Examples

```
WRITE £10
WRITE £10:47
```

Associated keywords

FETCH, RECORD, REC. See fetch-statement, record-statement, rec-call.

6. FUNCTIONS

This chapter defines the syntax and action of all the predefined functions available in IS-BASIC.

Each function description is split into two parts:

- name and value
- syntax

In the descriptions below, N, X, Y and Z stand for numeric-expressions. X$, Y$ and Z$ stand for string-expressions. V stands for a numeric-variable. V$ stands for a string-variable. A stands for a numeric- or string-array.

Each numeric-function accepts numeric arguments in any range except where noted. Where appropriate, numeric arguments are in the current angle unit (see option-statement).

ABS(X) - The absolute value of X.

abs-call = 'ABS'

ACOS(X) - The arccosine of X.

acos-call = 'ACOS'

ANGLE(X,Y) - The angle between the positive x-axis and the vector joining the origin to the point with co-ordinates (X,Y), where -PI < ANGLE(X,Y) ≤ PI. ANGLE(0,0) is 0. Note that anti-clockwise is positive.

angle-call = 'ANGLE'

ASIN(X) - The arcsine of X, where -PI/2 ≤ ASIN(X) ≤PI/2. -1 ≤ X ≤ 1.

asin-call = 'ASIN'

ATN(X) - The arctangent of X, where -(PI/2) < ATN(X) < (PI/2).

atn-call = 'ATN'

BIN(X) - The decimal equivalent of X, regarding the 1 and 0 characters in X as a binary number. The result is unpredictable if X contains digits other than 1 and 0.

bin-call = 'BIN'

BLACK - 0.

black-call = 'BLACK'

BLUE - 36.

blue-call = 'BLUE'

CEIL(X) - The smallest integer not less than X.

ceil-call = 'CEIL'

CHR$(X) - The one-characeter string containing the character with ordinal position X in the ASCII character set.

chr-call = 'CHR$'

COS(X) - The cosine of X.

cos-call = 'COS'

COSH - The hyperbolic cosine of X.

cosh-call = 'COSH'

COT(X) - The cotangent of X.

cot-call = 'COT'

CSC(X) - The cosecant of X.

csc-call = 'CSC'

CYAN - 182.

cyan-call = 'CYAN'

DATE$ - The current value of the internal date counter in the form specified by ANSI Standard X3.30 ie. "YYYYMMDD".

date-call = 'DATE$'

DEG(X) - The number of degrees in X radians.

deg-call = 'DEG'

EOF(X) - True if the end of file on channel X has been reached. Used with BASICs random record I/O statements.

EPS(X) - The smallest number representable in IS-BASIC which, when added to or subtracted from X will change the 12th or smaller digit of X.

eps-call = 'EPS'

EXLINE - The line number on which the last exception occured, or 0 if the last exception was from 'immediate mode'.

exline-call = 'EXLINE'

EXP(X) - The exponential of X ie. the value of the base of natural logarithms (e=2.7182818459) raised to the power X.

exp-call = 'EXP'

EXSTRING$(X) - A string containing the error message that would be printed by BASIC is exception number X occured when no exception was active.

exstring-call = 'EXSTRING$'

EXTYPE - The number of the last exception that occured.

extype-call = 'EXTYPE'

FREE - The number of bytes of RAM that are free for use by the current program.

free-call = 'FREE'

FP(X) - The fractional part of X ie. X-IP(X).

fp-call = 'FP'

GREEN - 146.

green-call = 'GREEN'

HEX$(X$) - A string of bytes which correspond to the hexadecimal numbers in X$. The hexadecimal numbers are separated by ',', and consist of one or two ASCII digits or upper- or lower-case characters in the range 'A' to 'F' or 'a' to 'f'.

hex-call = 'HEX$'

IN(port) - The byte read from the Z80 port.

in-call = 'IN'

INKEY$ - A string containing a single character read from the keyboard, or a string with zero length if no key was pressed.

inkey-call = 'INKEY$'

INF - The largest positive number representable by IS-BASIC.

inf-call = 'INF'

INT(X) - The largest integer not greater than X.

int-call = 'INT'

IP(X) - The integer part of X ie. SGN(X)*INT(ABS(X)).

ip-call = 'IP'

JOY(X) - The value read from joystick X.

joy-call = 'JOY'

LBOUND(A)
LBOUND(A,N) - The minimum value allowed for the Nth subscript of A. If N is omitted, then A must be a single-dimension array.

lbound-call = 'LBOUND'

LCASE$(X$) - The string resulting from the replacement off each upper-case ASCII character in X$ by its lower-case equivalent.

lcase-call = 'LCASE$'

LEN(X$) - The number of characters in X$.

len-call = 'LEN'

LOG(X) - The natural logarithm of X. X must be graeter than zero.

log-call = 'LOG'

LOG10(X) - The common logarithm of X. X must be greater than zero.

log10-call = 'LOG2'

LOG2(X) - The base 2 logarithm of X. X must be greater than zero.

log2-call = 'LOG2'

LTRIM$(X$) - X$ with all leading space characters removed.

ltrim-call = 'LTRIM$'

MAGENTA - 109.

magenta-call = 'MAGENTA'

MAX(X,Y) - The maximum value of X and Y.

max-call = 'MAX'

MAXLEN(V$) - The maximum number of characters that may be contained in V$.

maxlen-call = 'MAXLEN'

MIN(X,Y) - The minimum value of X and Y.

min-call = 'MIN'

MOD(X,Y) - X modulo Y ie. X-Y*INT(X/Y).

mod-call = 'MOD'

ORD(X$) - The ASCII value of the first characeter in X$.

ord-call = 'ORD'

PEEK(address) - The byte at the Z80 address.

peek-call = 'PEEK'

PI - 3.1415926536.

pi-call = 'PI'

POINTER(X) - The current file pointer for the file on channel X.

pointer-call = 'POINTER'

POS(X$,Y$)
POS(X$,Y$,N) - The character position within X$ of the first character of the first occurance of Y$ within X$, starting at character number N, or character number 1 if N is not given. If Y$ does not occure within the specified portion of X$, or if N is greater than the number of characters in X$, then zero is returned. If Y$ has a length of zero, then the value returned is N (or one if N is not given).

pos-call = 'POS'

RAD(X) - The number of radians in X degrees.

rad-call = 'RAD'

REC(X) - The current record number for channel X.

rec-call = 'REC'

RED - 73.

red-call = 'RED'

REM(X,Y) - The remainder of X divided by Y ie. X-Y*IP(X/Y).

rem-call = 'REM'

RGB(X,Y,Z) - The Enterprise absolute colour number for a colour consisting of X, Y and Z amounts of red, greem and blue respectively. $0 \leq X,Y,Z \leq 1$.

RND
RND(X) - A pseudo-random number. If X is not given, then the pseudo-random number returned is $0 \leq r < 1$, where r is the number returned. If X is given, then the number returned is a positive integer less then X. X must be in the range 1 to 32767.

rnd-call = 'RND'

ROUND(X,N) - The value of X rounded to N decimal digits to the right of the decimal point (or -N digits to the left if N < 0), ie. INT(X*10^N+0.5)/10^N.

round-call = 'ROUND'

RTRIM$(X$) - X$ with all trailing space characters removed.

rtrim-call = 'RTRIM$'

SEC - The secant of X.

sec-call = 'SEC'

SIN(X) - The sine of X.

sin-call = 'SIN'

SINH(X) - The hyperbolic sine of X.

sinh-call = 'SINH'

SIZE(A)
SIZE(A,N) - The number of permissible values for the Nth subscript of A, or the total number of elements in A if N is omitted.

size-call = 'SIZE'

SGN(X) - The sign of X: -1 if X < 0, 0 if X = 0, and +1 if X > 0.

sgn-call = 'SGN'

SPEEK(X,address) - The byte at address within segment X. Address is ANDed down to an offset within the segment, and X must be in the range 0 to 255 and is truncated to an integer.

speek-call = 'SPEEK'

STR$(X) - The string of ASCII characters that would be printed in a print-statement if X was printed, but without any leading spaces.

str-call = 'STR$'

SQR(X) - The square root of X. X must be positive.

sqr-call = 'SQR'

TAN(X) - The tangent of X.

tan-call = 'TAN'

TANH(X) - The hyperbolic tangent of X.

tanh-call = 'TANH'

TIME$ - The current value of the internal time counter in the format specified by ANSI Standard X3.43 ie. "HH:MM:SS".

time-call = 'TIME$'

TRUNCATE(X,N) - The value of X truncated to N decimal digits to the right of the decimal point (or -N digits to the left if N < 0), ie. IP(X*10^N)/10^N.

truncate-call = 'TRUNCATE'

UBOUND(A)
UBOUND(A,N) - The maximum value allowed for the Nth subscript of A. If N is omitted, then A must be a single-dimension array.

ubound-call = 'UBOUND'

UCASE$(X$) - The string resulting from the replacement off each lower-case ASCII character in X$ by its upper-case equivalent.

ucase-call = 'UCASE$'

USR(address)
USR(address,Y) - The value returned in Z80 register HL from the machine-code subroutine at address. Y, if given, is in HL when the subroutine is entered.

usr-call = 'USR'

VAL(X$) - The value of the numeric constant represented by the characters in X$ from the first character upto and including the last ASCII digit.

val-call = 'VAL'

VER$ - A string containing the version number and a copyright message.

ver-call = 'VER$'

VERNUM - The version number, 3.0.

vernum-call = 'VERNUM'

WHITE - 255.

white-call = 'WHITE'

YELLOW - 219.

yellow-call = 'YELLOW'

WORD$(X) - A two-character string containing the least-significant byte and the most-significant byte of X, which must be in the range 0 to 65535. The first character in the string is the most-significant byte.

word-call = 'WORD$'

