Enhanced BASIC language reference

Numbers

Numbers may range from zero to plus or minus 1.70141173x10^38 and will have an accuracy of just under 1 part in 1.68 x 10^7.

Numbers can be preceded by a sign, + or -, and are written as a string of numeric digits with or without a decimal point and can also have a positive or negative exponent as a power of 10 multiplier e.g.

-142 96.3 0.25 -136.42E-3 -1.3E7 1

.. are all valid numbers.

Integer numbers, i.e. with no decimal fraction or exponent, can also be in either hexadecimal or binary. Hexadecimal numbers should be preceded by **$** and binary numbers preceded by **%**, e.g.

%101010 -$FFE0 $A0127BD -%10011001 %00001010 $0A

.. again are all valid numbers.

Strings

Strings are any string of printable characters enclosed in a pair of quotation marks. Non printing characters may be converted to single character strings using the CHR$() functions.

"Hello world" "-136.42E-3" "+----+----+" "[Y/n]" "Y"

Are all valid strings.

Variables

Variables of both numeric and string type are available. String variables are distinguished by the **$** suffix. As well as simple variables arrays are also available and these may be either numeric or string and are distinguished by their bracketed indices after the variable name.

Variable names may be any length but only the first two name characters are significant so BL and BLANK will refer to the same variable. The first character must be one of "A" to "Z" or "a" to "z", following characters may also include numbers. E.g.

A A$ NAME$ x2LIM y colour s1 s2

Variable names are case sensitive so AB, Ab, aB and ab are all separate variables.

Variable names may not contain BASIC keywords. Keywords are only valid in upper case so 'PRINTER' is not allowed (it would be interpreted as PRINT ER) but 'printer' is.

Note that spaces in variable names are ignored so 'print e r', 'print er' and 'pri nter' will all be interpreted the same way.

BASIC Keywords

Here is a list of BASIC keywords. They are only valid when entered in upper case as shown and spaces may not be included in them. So GOTO is a valid BASIC keyword but GO TO is not. Click ▲ to return to keyword index.

ABS	AND	ASC	ATN	BIN$	BITCLR	BITSET
BITTST	CALL	CHR$	CLEAR	CONT	COS	DATA
DEC	DEEK	DEF	DIM	DO	DOKE	ELSE
END	EOR	EXP	FN	FOR	FRE	GET
GOSUB	GOTO	HEX$	IF	INC	INPUT	INT
IRQ	LCASE$	LEFT$	LEN	LET	LIST	LOAD
LOG	LOOP	MAX	MID$	MIN	NEW	NEXT
NMI	NOT	NULL	OFF	ON	OR	PEEK
PI	POKE	POS	PRINT	READ	REM	RESTORE
RETIRQ	RETNMI	RETURN	RIGHT$	RND	RUN	SADD
SAVE	SGN	SIN	SPC(	SQR	STEP	STOP
STR$	SWAP	TAB(	TAN	THEN	TO	TWOPI
UCASE$	UNTIL	USR	VAL	VARPTR	WAIT	WHILE
WIDTH	+	-	*	/	^	<<
>>	>	=	<			

- Anything in upper case is part of the command/function structure and must be present
- Anything in lower case enclosed in < > is to be supplied by the user
- Anything enclosed in [ ] is optional
- Anything enclosed in { } and separated by | characters are multi choice options
- Any items followed by an ellipsis, ... , may be repeated any number of times
- Any punctuation and symbols, except those above, are part of the structure and must be included

var	is a valid variable name
var$	is a valid string variable name
var()	is a valid array name
var$()	is a valid string array name

expression	is any expression returning a result
expression$	is any expression returning a string result

addr	is an integer in the range +/- 16777215 that will be wrapped to the range 0 to 65535
b	is a byte value 0 to 255
n	is an integer in the range 0 to 63999
w	is an integer in the range -32768 to 32767
i	is a positive integer value

r	is real number
+r	is a positive value real number (0 is considered positive)
$	is a string literal

BASIC Commands

BITCLR <addr>,<b>

Clears bit b of address addr. Valid bit numbers are 0, the least significant bit, to 7, the most significant bit. Values outside this range will cause a function call error. ▲

BITSET <addr>,<b>

Sets bit b of address addr. Valid bit numbers are 0, the least significant bit, to 7, the most significant bit. Values outside this range will cause a function call error. ▲

CALL <addr>

CALLs a user subroutine at address addr. No values are passed or returned and so this is much faster than using USR(). See extending CALL for details. ▲

CLEAR

Erases all variables and functions and resets FOR .. NEXT, GOSUB .. RETURN and DO ..LOOP states. ▲

CONT

Continues program execution after CTRL-C has been typed, a STOP has been encountered during program execution or a null input was given to an INPUT request. ▲

DATA [{r|$}[,{r|$}]...]

Defines a constant or series of constants. Real constants are held as strings in program memory and can be read as numeric values or string values. String constants may contain spaces but if they need to contain commas then they must be enclosed in quotes. ▲

DEC <var>[,var]...

Decrement variables. The variables listed will have their values decremented by one. Trying to decrement a string variable will give a type mismatch error. DEC A is much faster than doing A=A-1 and DEC A,A is slightly faster than doing A=A-2. ▲

DEF FN <name>(<var>) = <statement>

Defines <statement> as function <name>. <name> can be any valid numeric variable name of one or more characters. <var> must be a simple variable and is used to pass a numeric argument into the function.

Note that the value of <var> will be unchanged if it is used in the function so <var> should be considered to be a local variable name. ▲

DIM <var[$](i1[,i2[,in]...])>[,var[$](i1[,i2[,in]...])]...

Dimension arrays. Creates arrays of either string or numeric variables. The arrays can have one or more dimensions. The lower limit of any dimension is always zero and the upper limit is i. If you do not explicitly dimension an array then it's number of dimensions will be set when you first access it and the upper bound will be set to 10 for each dimension. ▲

DO

Marks the beginning of a DO .. LOOP loop (See LOOP). No parameters. This command can be nested like FOR .. NEXT or GOSUB .. RETURN. ▲

DOKE <addr,w>

Writes the word value w into the addresses addr and addr+1, the lower byte of w is in addr. Note if addr = 65535 ($FFFF) then the high byte will be written to address zero. ▲

ELSE

See IF. ▲

END

Terminates program execution and returns control to the command line (direct mode). END may be placed anywhere in a program, it does not have to be on the last line, and there may be any number, including none, of ENDs in total.

Note. CONT may be used after and END to resume execution from the next statement. ▲

FN<name>(<expression>)

See DEF. ▲

FOR <var> = <expression> TO <expression> [STEP expression]

Assigns a variable to a loop counter and optionally sets the start value, the end value and the step size. If STEP expression is omitted then a default step size of +1 will be assumed. ▲

GET <var[$]>

Gets a key, if there is one, from the input device. If there is no key waiting then var will be set to 0 and var$ will return a null string "". GET does not halt and execution will continue. ▲

GOSUB <n>

Call a subroutine at line n. Program execution is diverted to line n but the calling point is remembered. Upon encountering a RETURN statement program execution will continue with the next statement (line) after the GOSUB. ▲

GOTO <n>

Continue execution from line number n. ▲

IF <expression> {GOTO<n>|THEN<{n|statement}>}[ELSE<{n|statement}>]

Evaluates expression. If the result of expression is non zero then the GOTO or the statement after the THEN is executed. If the result of expression is zero then execution continues with the next line.

If the result of expression is zero and the optional ELSE clause is included then the statement after the ELSE is executed.

IF .. THEN .. ELSE .. behaves as a single statement so in the line ..

```
IF <expression> THEN <statement one> ELSE <statement two> :
<statement three>
```

.. statement three will always be executed regardless of the outcome of the IF as long as the executed statement was not a GOTO. ▲

INC <var>[,var]...

Increment variables. The variables listed will have their values incremented by one. Trying to increment a string variable will give a type mismatch error. INC A is much faster than doing A=A+1 and INC A,A is slightly faster than doing A=A+2. ▲

INPUT ["$";] <var>[,var]...

Get a variable, or list of variables from the input stream. A question mark, "?", is always output, after the string if there is one, and if further input is required, i.e. there are more variables in the list than the user entered values, then a double question mark, "??", will be output until enough values have been entered.

There are two possible messages that may appear during the execution of an input statement:

Extra ignored

The user has attempted to enter more values than are required. Program execution will continue but the extraneous data entered has been discarded.

Redo from start

The user has attempted to enter a string where a number was expected. The reverse never causes an error as numbers are also valid strings. ▲

IRQ {ON|OFF|CLEAR}

Enables or disables the IRQ handling subroutine. Note that turning the handler off does not suppress the interrupt detection and if an interrupt occurs while handling is off it will be

actioned as soon as handling is turned back on. Using CLEAR clears the interrupt assignment and it can only be restarted with an ON IRQ command. ▲

LET <var> = <expression>

Assign the value of expression to var. Both var and expression bust be of the same type. The LET command word is optional and just <var> = <expression> will give exactly the same result. It is only maintained for historical reasons. ▲

LIST [n1][-n2]

Lists the entire program held in memory. If n1 is specified then the listing will start from line n1 and run to the end of the program. If -n2 is specified then the listing will terminate after line n2 has been listed. If n1 and -n2 are specified then all the lines from n1 to n2 inclusive will be listed.

Note. If n1 does not exist then the list will start from the next line numbered after n1. If n2 does not exist then the listing will stop with the last line numbered before n2.

Also note. LIST can be executed from within a program, first a [CR][LF] is printed and then the specified lines, if any, each terminated with another [CR][LF]. Program execution then continues as normal. ▲

LOAD

Does nothing in this version but does it via a vector in RAM so is easily patched. ▲

LOOP [{UNTIL|WHILE} expression]

Marks the end of a DO .. LOOP loop. There are three possible variations on the LOOP command ..

> **LOOP**
>
> This loop repeats forever. With just this command control is passed back to the next command after the corresponding DO.
>
> **LOOP UNTIL expression**
>
> This loop will repeat until the value of expression is non zero. Once that occurs execution will continue with the next command after the LOOP UNTIL.
>
> **LOOP WHILE expression**
>
> This loop will repeat while the value of expression is non zero. When the value of expression is zero execution will continue with the next command after the LOOP WHILE. ▲

NEW

Deletes the current program and all variables from memory. ▲

NEXT [var[,var]...]

Increments or decrements a loop variable and checks for the terminating condition. If the terminating condition has been reached then execution continues with the next command, else execution continues with the command after the FOR assignment. *See FOR*. ▲

NMI {ON|OFF|CLEAR}

Enables or disables the NMI handling subroutine. Note that turning the handler off does not suppress the interrupt detection and if an interrupt occurs while handling is off it will be actioned as soon as handling is turned back on. Using CLEAR clears the interrupt assignment and it can only be restarted with an ON NMI command. ▲

NOT <expression>

Generates the bitwise NOT of then signed integer value of <expression>. ▲

NULL <n>

Sets the number of null characters printed by BASIC after every carriage return. n may be specified in the range 0 to 255. ▲

OFF

See IRQ or NMI. ▲

ON <expression> {GOTO|GOSUB} <n>[,n]...

The integer value of expression is calculated and then the nth number after the GOTO or GOSUB is taken (where n is the result of expression). Note that valid results for expression range only from zero to 255. Any result outside this range will cause a Function call error. ▲

ON {IRQ|NMI} <n>

Set up the IRQ or NMI routine pointers. This sets up the effective GOSUB line that is taken when an interrupt happens. When the effective GOSUB is taken the interrupt, IRQ or NMI, is turned off. This can be turned back on with the interrupt on command or by using the matching special return. The normal program flow is resumed by any of RETIRQ, RETNMI or RETURN. ▲

POKE <addr,b>

Writes the byte value b into the address addr. ▲

PRINT [expression][{;|,}expression]...[{;|,}]

Outputs the value of each expressions. If the list of expressions to be output does not end with a comma or a semi-colon, then a carriage return and linefeed is output after the values.

Expressions on the line can be separated with either a semi-colon, causing the next expression to follow immediately, or a comma which will advance the output to the next tab stop before continuing to print. If there are no expressions and no comma or semi-colon after the PRINT statement then a carriage return and linefeed is output.

When entering a program line, or immediate statement, PRINT can be abbreviated to ? ▲

READ <var>[,var]...

Reads values from DATA statements and assigns them to variables. Trying to read a string literal into a numeric variable will cause a syntax error. ▲

REM

Everything following this statement on this program line will be ignored, even colons. ▲

RESTORE [n]

Reset the DATA pointer. If n is specified then the pointer will be reset to the beginning of line n else it will be reset to the start of the program. If n is specified but doesn't exist an error will be generated. ▲

RETIRQ

Returns program execution to the next statement after an interrupt, automatically restores the IRQ enabled flag. See ON IRQ. ▲

RETNMI

Returns program execution to the next statement after an interrupt, automatically restores the NMI enabled flag. See ON NMI. ▲

RETURN

Returns program execution to the next statement (line) after the last GOSUB encountered. See GOSUB. Also returns program execution to the next statement after an interrupt but does not restore the enabled flags. ▲

RUN [n]

Begins execution of the program currently in memory at the lowest numbered line. RUN erases all variables and functions, resets FOR .. NEXT, GOSUB .. RETURN and DO ..LOOP states and sets the data pointer to the program start.

If n is specified then programme execution will start at the specified line number. ▲

SAVE

Does nothing in this version but does it via a vector in RAM so is easily patched. ▲

SPC(<expression>)

Prints <expression> spaces. This command is only valid in a PRINT statement. ▲

STEP

Sets the step size in a FOR .. NEXT loop. See FOR. ▲

STOP

Halts program execution and generates a "Break in line n" message where n is the line in which the STOP was encountered. ▲

SWAP <var[$]>,<var[$]>

Swap two variables. The variables listed will have their values exchanged. Both must be of the same type, numeric or string, and either, or both, may be array elements. Trying to swap a numeric and string variable will give a type mismatch error. ▲

TAB(<expression>)

Sets the cursor position to <expression>. If the cursor is already beyond that point then the cursor will be left where it is. This command is only valid in a PRINT statement. ▲

THEN

See IF. ▲

TO

Sets the range in a FOR .. NEXT loop. See FOR. ▲

UNTIL

See DO and LOOP. ▲

WAIT <addr,b1>[,b2]

Program execution will wait at this point until the value of the location addr exclusive ORed with b2 then ANDed with b1 is non zero. If b2 is not defined then it is assumed to be zero. Note b1 and b2 must both be byte values. ▲

WHILE

See DO and LOOP. ▲

WIDTH {b1|,b2|b1,b2}

Sets the terminal width and TAB spacing. b1 is the terminal width and b2 is the tab spacing, default is 80 and 14. Width can be zero, for "infinite" terminal width, or from 16 to 255. The tab size is from 2 to width-1 or 127, whichever is smaller. ▲

BASIC Operators

Operators perform mathematical or logical operations on values and return the result. The operation is usually preceded by a variable name and equality sign or is part of an IF .. THEN statement.

+	Add. c = a + b will assign the sum of a and b to c.
-	Subtract. c = a - b will assign the result of a minus b to c.
*	Multiply. c = a * b will assign the product of a and b to c.
/	Divide. c = a / b will assign the result of a divided by b to c.
^	Raise to the power of. c = a ^ b will assign the result of a raised to the power of b to c.
AND	Logical AND. c = a AND b will assign the logical AND of a and b to c
EOR	Logical Exclusive OR. c = a EOR b will assign the logical exclusive OR of a and b to c.
OR	Logical OR. c = a OR b will assign the logical inclusive OR of a and b to c.
<<	Shift left. c = a << b will assign the result of a shifted left by b bits to c.
>>	Shift right. c = a >> b will assign the result of a shifted right by b bits to c.
=	Equals. c = a = b will assign the result of the comparison a = b to c.
>	Greater than. c = a < b will assign the result of the comparison a > b to c.
<	Less than. c = a < b will assign the result of the comparison of a < b to c.

The three comparison operators can be mixed to provide further operators ..

>= or =>	Greater than or equal to.
<= or =<	Less than or equal to.
<> or ><	Not equal to (greater than or less than).
<=> any order	Always true (greater than or equal to or less than). ▲

BASIC Functions

Functions always return a value, be it numeric or string, so are used on the right hand side of the = sign in assignments, on either side of operators and in commands requiring an expression e.g. after PRINT, within expressions, or in other functions.

ABS(<expression>)

Returns the absolute value of <expression>. ▲

ASC(<expression$>)

Returns the ASCII value of the first character of <expression$>. ▲

ATN(<expression>)

Returns, in radians, the arctangent of <expression>. ▲

BIN$(<expression>[,b])

Returns <expression> as a binary string. If b is omitted, or if b = 0, then the string is returned with all leading zeroes removed and is of variable length. If b is set, permissible values range from 1 to 24, then a string of length b will be returned. The result is always unsigned and calling this function with expression > 2^24-1 or b > 24 will cause a function call error. ▲

BITTST(<addr>,<b>)

Tests bit b of address addr. Valid bit numbers are 0, the least significant bit, to 7, the most significant bit. Values outside this range will cause a function call error. Returns zero if the bit was zero, returns -1 if the bit was 1. ▲

COS(<expression>)

Returns the cosine of the angle <expression> radians. ▲

DEEK(<addr>)

Returns the word value of <addr> and addr+1 as an integer in the range -32768 to 32767. Addr holds the word low byte. ▲

EXP(<expression>)

Returns e^<expression>. Natural antilog. ▲

FRE(<expression>)

Returns the amount of free program memory. The value of expression is ignored and can be numeric or string. ▲

HEX$(<expression>[,b])

Returns <expression> as a hex string. If b is omitted, or if b = 0, then the string is returned with all leading zeroes removed and is of variable length. If b is set, permissible values range from 1 to 6, then a string of length b will be returned. The result is always unsigned and calling this function with expression > 2^24-1 or b > 6 will cause a function call error. ▲

INT(<expression>)

Returns the integer of <expression>. ▲

LCASE$(<expression$>)

Returns <expression$> with all the alpha characters in lower case. ▲

LEFT$(<expression$,b>)

Returns the leftmost b characters of <expression$>. ▲

LEN(<expression$>)

Returns the length of <expression$>. ▲

LOG(<expression>)

Returns the natural logarithm (base e) of <expression>. ▲

MAX(<expression>[,<expression>]...)

Returns the maximum value from a list of numeric expressions. There must be at least one expression but the upper limit is dictated by the line length. Each expression is evaluated in turn and the largest of them returned. ▲

MID$(<expression$,b1>[,b2])

Returns the substring string from character b1 of expression$ of length b2. The characters of expression$ are numbered from 1 starting with the leftmost. If b2 is omitted then all the characters from b1 to the end of the string are returned. ▲

MIN(<expression>[,<expression>]...)

Returns the minimum value from a list of numeric expressions. There must be at least one expression but the upper limit is dictated by the line length. Each expression is evaluated in turn and the smallest of them returned. ▲

PEEK(<addr>)

Returns the byte value of <addr>. ▲

PI

Returns the value of pi as 3.14159274 (closest floating value). ▲

POS(<expression>)

Returns the POSition of the cursor on the terminal line. The value of expression is ignored. ▲

RIGHT$(<expression$,b>)

Returns the rightmost b characters of <expression$>. ▲

RND(<expression>)

Returns a random number in the range 0 to 1. If the value of <expression> is non zero then it will be used as the seed for the returned pseudo random number otherwise the next number in the sequence will be returned. ▲

SADD(<{var$|var$()}>)

Returns the address of var$ or var$(). This returns a pointer to the actual string in memory not the descriptor. If you want the pointer to the descriptor use VARPTR instead. ▲

SGN(<expression>)

Returns the sign of <expression>. If the value is positive SGN returns +1, if the value is negative then SGN returns -1. If expression=0 then SGN returns 0. ▲

SIN(<expression>)

Returns the sine of the angle <expression> radians. ▲

SQR(<expression>)

Returns the square root of <expression>. ▲

STR$(<expression>)

Returns the result of <expression> as a string. ▲

TAN(<expression>)

Returns the tangent of the angle <expression> radians. ▲

TWOPI

Returns the value of 2*pi as 6.28318548 (closest floating value). ▲

UCASE$(<expression$>)

Returns <expression$> with all the alpha characters in upper case. ▲

CHR$(b)

Returns single character string of character <b>. ▲

USR(<expression>)

Takes the value of <expression> and places it in FAC1 and then calls the USeR routine pointed to by the vector at $0B,$0C. What the routine does with this value is entirely up to the user, it can even be safely ignored if it isn't needed. The routine, after the user code has done an RTS, takes whatever is in FAC1 and returns that. Note it can be either a numeric or string value. See using USR() for details.

If no value needs to be passed or returned then CALL is a better option. ▲

VAL(<expression$>)

Returns the value of <expression$>. ▲

VARPTR(<var[$]>)

Returns a pointer to the variable memory space. If the variable is numeric, or a numeric array element, then VARPTR returns the pointer to the packed value of that variable in memory. If the variable is a string, or a string array element, then VARPTR returns a pointer to the descriptor for that string. If you want the pointer to the string itself use SADD instead. ▲

BASIC Error Messages

These all occur from time to time and, if the error occurred while executing a program, will be followed by "in line " where is the number of the line in which the error occurred.

Array bounds Error

An attempt was made to access an element of an array that was outside it's bounding dimensions.

Can't continue Error

Execution can't be continued because either the program execution ended because an error occurred, NEW or CLEAR have been executed since the program was interrupted or the program has been edited.

Divide by zero Error

The right hand side of an A/B expression was zero.

Double dimension Error

An attempt has been made to dimension an already dimensioned array. This could be because the array was accessed previously causing it to be dimensioned by default.

Function call Error

Some parameter of a function was outside it's limits. E.g. Trying to POKE a value of less than 0 or greater than 255.

Illegal direct Error

An attempt was made to execute a command or function in direct mode which is disallowed in that mode e.g. INPUT or DEF.

LOOP without DO Error

LOOP has been encountered and no matching DO could be found.

NEXT without FOR Error

NEXT has been encountered and no matching FOR could be found.

Out of DATA Error

A READ has tried to read data beyond the last item. Usually because you either mistyped the DATA lines, miscounted the DATA, RESTOREd to the wrong place or just plain forgot to restore.

Overflow Error

The result of a calculation has exceeded the numerical range of BASIC. This is plus or minus 1.7014117+E38

Out of memory Error

Anything that uses memory can cause this but mostly it's writing and running programmes that does it.

RETURN without GOSUB Error

RETURN has been encountered and no matching GOSUB could be found.

String too complex Error

A string expression caused an overflow on the descriptor stack. Try splitting the expression into smaller pieces.

String too long Error

String lengths can be from zero to 255 characters, more than that and you will see this.

Syntax Error

Just generally wrong. 8^)=

Type mismatch Error

An attempt was made to assign a numeric value to a string variable, a string value to a numeric variable or a value of one type was returned when a value of the other type was expected or an attempt at a relational operation between a string and a number was made.

Undefined function Error

FN <var> was called but not found.

Undefined statement Error

Either a GOTO, GOSUB, RUN or RESTORE was attempted to a line that doesn't exist or the line referred to in an ON <expression> {GOTO|GOSUB} or ON {IRQ|NMI} doesn't exist.