



## Language reference. By Lee Davison.



### Numbers

Floating numbers may range from zero to plus or minus  $1.70141173 \times 10^{38}$  and will, when assigned as variables, have an accuracy of just under 1 part in  $1.68 \times 10^7$ . During calculations the mantissa is extended to 32 bits with a corresponding increase in accuracy.

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 may range from -2147483648 to 4294967295 but numbers above 2147483647 will 'wrap round' to their equivalent 2s complement negative value. Integers can also be expressed 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 floating point, integer and string type are available. String variables are distinguished by the \$ suffix, integer variables are distinguished by the & suffix. As well as simple variables arrays are also available and these may be floating point, integer or string and are distinguished by their bracketed indices after the variable name.

Variable names may be any length but only the first four name characters are significant. The first character must be one of "A" to "Z" or "a" to "z", any other 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 valid but GO TO is not.

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

- 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 -2147483648 to 4294967295 that will be wrapped to the range 0 to 4294967295

b is a byte value from 0 to 255

n is a positive integer in the range 0 to 2147483647

w is an integer in the range -32768 to 32767

uw is a positive integer in the range 0 to 65535

l is an integer in the range -2147483648 to 4294967295 that will be wrapped to the range -2147483648 to 2147483647

i is a +ve integer value

r is real number

+r is a +ve value real number (0 is considered +ve)

\$ is a string literal

## **BASIC Commands**

### **END**

Terminates program execution and returns control to the command line (direct mode). END may be placed anywhere in a program and any number, including none, is allowed.

Note. [CONT](#) may be used after and END to resume execution from the next statement.

### **FOR <var[&]> = <expression> TO <expression> [STEP expression]**

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

### **NEXT [var[,var]...]**

Increments 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.

### **DATA [{r|\$}]L,{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.

### **INPUT ["prompt\$";] <var>[,var]...**

Get a variable, or list of variables from the input stream. A question mark, "?", is always output (after the prompt 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.

Note that the prompt string, **prompt\$**, must always be a string enclosed in quotes and can not be a string variable or string expression.

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.

### **DIM <var[{\$|&}](i1[,i2[,i3]])>[,var[{\$|&}](i1[,i2[,i3]])]...**

Dimension arrays. Creates arrays of either string, integer or floating variables. The arrays can have one, two or three dimensions. The lower limit is always zero and the

upper limit is i. If you do not dimension an array then when you first try to access it an 'Undimensioned array' error will be generated.

The array created by the DIM statement will have all it's elements zeroed on creation.

### **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.

### **LET <var> = <expression>**

Assign the value of expression to var. Both var and expression must be of the same basic type, i.e. both numeric or both string. The LET command word is optional and just <var> = <expression> will give exactly the same result. It is only maintained for historical reasons.

### **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.

### **SWAP <var[{\$|&}]>,<var[{\$|&}]>**

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

### **GOTO <n>**

Continue execution from line number n

### **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.

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

Evaluates expression. If the result of expression is non zero then the statement(s) after the after the expression are executed. If the result of expression is zero then execution continues with the next line.

Note. There are three valid ways to write this command ..

**IF <expression> GOTO <n>**

**IF <expression> THEN <n>**

**IF <expression> THEN <statement>**

**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.

**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.

**RETURN**

Returns program execution to the next statement (line) after the last [GOSUB](#) encountered.

**REM**

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

**STOP**

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

**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). If the result of <expression> is zero, or greater than the number of destination line numbers, then execution will continue with the instruction following this one.

Note that valid results for expression range only from zero to 255. Any result outside this range will cause a 'Function call' error.

### **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.

### **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.

### **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.

### **LOAD <expression\$>**

Usually does nothing but does it via a vector in RAM so is easily patched.

For the EASy68k simulator this attempts to load the file expression\$. It will recognise the file format (binary or ASCII) and invoke the necessary routines.

During an ASCII load you will see the program as it is being read from the file. This file is also merged with any program lines currently in the memory.

Binary loading is *much* faster and over writes any program currently in memory.

### **SAVE <expression\$>[,a[n1]][-n2]]**

Usually does nothing but does it via a vector in RAM so is easily patched.

For the EASy68k simulator this attempts to save the program memory to the file expression\$. If the command has the ,a appended then the program will be saved in ASCII format. If saving in ASCII format then the range of the save can be specified as with the [LIST](#) command.

### **DEF FN <name[{\$|&}]>(<var[{\$|&}]>) = <statement>**

Defines <statement> as function <name>. <name> can be any valid variable name, of any type, of one or more characters. <var> can be any simple variable (i.e. not an array element), again of any type, and is used to pass the function argument into the function.

Note that the value of <var> will be unchanged by it's use in the function so <var> should be considered to be a local variable name.

### **POKE <addr,b>**

Writes the byte value b into the address addr.

### **DOKE <addr,w>**

Writes the word value w into the addresses addr and addr+1, the upper byte of w is in addr.

**Note:** addr must be an even integer or an 'address' error will be generated.

### **LOKE <addr,l>**

Writes the longword value l into the addresses addr, addr+1, addr+2, and addr+3. The most significant byte of l is in addr.

**Note:** addr must be an even integer or an 'address' error will be generated.

### **CALL <addr>**

CALLs a user subroutine at address addr. No values are passed or returned and so this is much faster than using [USR\(\)](#).

### **DO**



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

### **LOOP [{UNTIL|WHILE} expression]**

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

#### **LOOP**

Repeats forever. With just this command control is passed back to the next command (line) after the 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 (line) after this one.

#### **LOOP WHILE expression**

This loop will repeat while the value of expression is non zero. Once expression is zero execution will continue with the next command (line) after this one.

### **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 ? .

### **CONT**

Continues program execution after CTRL-C has been typed, a [STOP](#) has been encountered during program execution, a null input was given to an [INPUT](#) request or [END](#) was reached.

If CONT is used to restart from an aborted INPUT command then the INPUT, along with any prompt string will be repeated.

### **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.

### **CLEAR**

Erases all variables and functions and resets [FOR](#) .. [NEXT](#), [GOSUB](#) .. [RETURN](#) and [DO](#) .. [LOOP](#) states.

### **NEW**

Deletes the current program and all variables from memory.

### **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.

### **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.

### **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. **TAB()** has an implied semi colon after it so there would be no **[CR][LF]** printed after **PRINT TAB(n)**, this is the same as with the [SPC\(\)](#) command. This command is only valid in a [PRINT](#) statement.

### **TO**

Sets the range in a [FOR](#) .. [NEXT](#) loop.

### **FN<name[{\$|&}]>(<expression[{\$|&}]>)**

Returns the result of **FNname()** for the value of **expression**. See [DEF](#) for more details.

### **SPC(<expression>)**

Prints <expression> spaces. **SPC()** has an implied semi colon after it so there would be no **[CR][LF]** printed after **PRINT SPC(n)**, this is the same as with the [TAB\(\)](#) command. This command is only valid in a [PRINT](#) statement.

### **THEN**

See [IF](#).

### **NOT <expression>**

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

### **STEP**

Sets the step size in a [FOR](#) .. [NEXT](#) loop.

### **UNTIL**

See [DO](#) and [LOOP](#).

### **WHILE**

See [DO](#) and [LOOP](#).

### **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.

## **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.  $c = a \text{ AND } b$  will assign the logical AND of *a* and *b* to *c*.
- EOR** Logical Exclusive OR.  $c = a \text{ EOR } b$  will assign the logical exclusive OR of *a* and *b* to *c*.
- OR** Logical OR.  $c = a \text{ OR } b$  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 ..

<code>&gt;=</code> or <code>=&gt;</code>	Greater than or equal to.
<code>&lt;=</code> or <code>=&lt;</code>	Less than or equal to.
<code>&lt;&gt;</code> or <code>&gt;&lt;</code>	Not equal to (greater than or less than).
<code>&lt;=&gt;</code> 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 or in commands requiring an expression e.g. after PRINT, within expressions, or in other functions.

### **SGN(<expression>)**

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

### **INT(<expression>)**

Returns the integer of <expression>.

### **ABS(<expression>)**

Returns the absolute value of <expression>.

### **USR(<expression>)**

Takes the value of <expression> and places it in FAC1 and then calls the USer routine pointed to by the USer vector. 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 floating, integer or string value.

If no value needs to be passed or returned then [CALL](#) is a better option.

### **FRE(<expression>)**

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

### **POS(<expression>)**

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

**SQR(<expression>)**

Returns the square root 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.

**LOG(<expression>)**

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

**EXP(<expression>)**

Returns  $e^{\text{expression}}$ . (natural antilog)

**COS(<expression>)**

Returns the cosine of <expression>.

**SIN(<expression>)**

Returns the sine of <expression>.

**TAN(<expression>)**

Returns the tangent of <expression>.

**ATN(<expression>)**

Returns the arctangent of <expression>.

**PEEK(<addr>)**

Returns the byte value of <addr>.

**DEEK(<addr>)**

Returns the word value of <addr> and addr+1. Addr holds the word high byte.

**Note:** addr must be an even integer or an 'address' error will be generated.

**LEEK(<addr>)**

Returns the longword value of `addr`, `addr+1`, `addr+2` and `addr+3`. `Addr` holds the most significant byte, `addr+3` the least significant byte.

**Note:** `addr` must be an even integer or an 'address' error will be generated.

### **SADD(<{var\$|var\$())|expression\$>)**

Returns the address of `var$`, `var$()` or `expression$`. This returns a pointer to the actual string in memory not the descriptor.

### **LEN(<expression\$>)**

Returns the length of `<expression$>`.

### **STR\$(<expression>)**

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

### **VAL(<expression\$>)**

Returns the value of `<expression$>`.

### **ASC(<expression\$>)**

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

### **LCASE\$(<expression\$>)**

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

### **UCASE\$(<expression\$>)**

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

### **CHR\$(b)**

Returns single character string of character `<b>`.

### **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 set values are 1 to 8) then a string of length `b` will be returned. The result is always unsigned and calling this

function with `expression>2^32-1` or `b>8` will cause a function call error.

### **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 set values are 1 to 32) then a string of length b will be returned. The result is always unsigned and calling this function with `expression>2^32-1` 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.

### **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 value of the largest of them 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 value of the smallest of them returned.

### **RAMBASE**

Returns the address of the start of EhBASIC's RAM.

### **PI**

Returns the value of pi as 3.14159265347 (closest floating value).

### **TWOPI**

Returns the value of 2\*pi as 6.28318530694 (closest floating value).



**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 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.

**LEFT\$(<expression\$,uw>)**

Returns the leftmost uw characters of <expression\$>. If uw=0 then a null string will be returned.

**RIGHT\$(<expression\$,uw>)**

Returns the rightmost uw characters of <expression\$>. If uw=0 then a null string will be returned.

**MID\$(<expression\$,uw1>[,uw2])**

Returns the sub string string from character uw1 of <expression\$> of length uw2. The characters of <expression\$> are numbered from 1 starting with the leftmost. If uw1=0 then a 'Function call' error will be generated. If uw1>string length or uw2=0 then a null string will be returned.

**BASIC Error Messages**

These will all be followed by the word Error and, if the error occurred while executing a program, will be followed by "in line <n>" where <n> is the number of the line in which the error occurred.

**NEXT without FOR**

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

**Syntax**

Just generally worng. 8^)=

**RETURN without GOSUB**

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

**Out of DATA**

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.

**Function call**

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.

**Overflow**

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

**Out of memory**

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

**Undefined statement**

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} doesn't exist.

**Array bounds**

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

**Double dimension**

An attempt has been made to dimension an already dimensioned array.

**Divide by zero**

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

**Illegal direct**

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

**Type mismatch**

An attempt was made to pass a numeric value to a string, a string value to a numeric variable or an attempt at a relational operation between a string and a number was made.

**String too long**

String lengths can be from zero to 65535 characters, more than that and you will see this.

**String too complex**

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

**Can't continue**

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.

**Undefined function**

FN <name[{\$|&}]> was called but not found.

**LOOP without DO**

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

**Undefined variable**

A variable has been referenced that has yet to have a value assigned to it.

**Undimensioned array**

Aa array has been referenced that has yet to be dimensioned.

**Wrong dimensions**

Aa array has been referenced with the incorrect number of dimensions in the reference.

**Address**

Some function that requires an even address as a parameter has been passed an odd address.