
BaCon BASIC converter documentation - version 1.0 build 16

Contents

- [Introduction](#)
 - [BaCon usage and parameters](#)
 - [General syntax](#)
 - [Mathematics, variables](#)
 - [Arrays](#)
 - [Associative arrays](#)
 - [Declaration](#)
 - [Relations, lookups](#)
 - [Basic logic programming](#)
 - [Strings by value or by reference](#)
 - [Execute BaCon source program using a 'shebang'](#)
 - [Creating and linking to libraries created with BaCon](#)
 - [Step 1: create a library](#)
 - [Step 2: compile the library](#)
 - [Step 3: copy library to a system path](#)
 - [Step 4: update linker cache](#)
 - [Step 5: demonstration program](#)
 - [Step 6: compile and link](#)
 - [Statements and functions](#)
 - [Appendix A: Runtime error codes](#)
-

Introduction

BaCon is an acronym for BAsic CONverter. The BaCon BASIC converter is a tool to convert programs written in BASIC syntax to C. The resulting C program should be compilable with GCC or CC.

BaCon is intended to be a programming aid in creating small tools which can be compiled on different Unix-based platforms. It tries to revive the days of the good old [BASIC](#).

The BaCon converter passes expressions and numeric assignments to the C compiler without verification or modification. This means that the assignment symbol '=', which is used in C programs, also must be used in BaCon programs. Another consequence is that the C comparison symbol '==' also must be used in BaCon (there are aliases for the comparison symbol like 'EQ' or 'IS').

Therefore BaCon can be considered a *lazy converter*: it relies on the expression parser of the C compiler.

BaCon usage and parameters

To use BaCon, download the converter and make sure the program has executable rights. There is no difference between the Kornshell version or the BASH version. Only one of them is needed. Suppose the BASH version is downloaded, the converter can be used as follows:

```
bash ./bacon.bash myprog.bac
```

By default the converter will refer to '/bin/bash' by itself. It uses a so-called '[shebang](#)' which allows the program to run standalone provided the executable rights are set correctly. This way there is no

need to execute BaCon with an explicit use of BASH. So this is valid:

```
./bacon.bash myprog.bac
```

All BaCon programs should use the '.bac' extension. But it is not necessary to provide this extension for conversion. So BaCon also understands the following syntax:

```
./bacon.bash myprog
```

The BaCon Basic Converter can be started with the following parameters.

- -c: determine which C compiler should create the binary. The default value is 'gcc'.
Example: ./bacon -c cc prog. In this situation, the converted program will be compiled by a C compiler called 'cc'.
- -l: pass libraries to the C linker
- -o: pass compiler options to the C compiler
- -i: the compilation will use an additional external C include file
- -d: determine the directory where BaCon should store the generated C files. Default value is the current directory.
- -f: create a shared object of the program
- -n: do not compile the C code automatically after conversion.
- -j: invoke C preprocessor to interpret C macros which were added to BaCon source code.
- -p: do not cleanup the generated C files. Default behavior is to delete all generated C files automatically.
- -b: use bacon in the 'shebang' so BaCon programs can be executed similar to a script (binary version of BaCon only).
- -v: shows the current version of BaCon.
- -h: shows an overview of all possible options on the prompt. Same as the '-?' parameter.

So how to pass compiler and linker flags to the C compiler? Here are a few examples.

- Convert and compile program with debug symbols: ./bacon -o "-g" yourprogram.bac
- Convert and compile program and export functions as symbols: ./bacon -o "-export-dynamic" yourprogram.bac
- Convert and compile program using TCC and export functions as symbols: ./bacon -c tcc -o "-rdynamic" yourprogram.bac
- Convert and compile program forcing 32bit and optimize for platform: ./bacon -o "-m32 -mtune=native" yourprogram.bac
- Convert and compile program linking to a particular library: ./bacon -l somelib yourprogram.bac

General syntax

BaCon consists of statements, functions and expressions. Each line should begin with a statement. A line may continue onto the next line by using the '\' symbol at the end. The [LET](#) statement may be omitted, so a line may contain an assignment. Functions always return a value or string. Functions created in the program can be used standalone. Expressions are not converted but are passed unchanged to the C compiler (*lazy conversion*).

BaCon does not need line numbers. One statement per line is accepted. If there are more statements on the same line then an error occurs.

All keywords must be written in capitals. Keywords in small letters are considered to be variables. Statements are always written without using brackets. Only functions must use brackets to enclose their arguments.

Variables will be declared implicitly from the first moment a variable is used. If a variable name ends with the '\$' symbol, a string variable is assumed. Otherwise it is considered to be numeric. By

default, BaCon assumes long type with NUMBER variables. With the ['DECLARE'](#) statement it is possible to define a variable to any other C-type explicitly.

The three main types in BaCon are defined as STRING, NUMBER and FLOATING. These are translated to char*, long and double.

Subroutines may be defined using [SUB/ENDSUB](#) and do not return a value. With the [FUNCTION/ENDFUNCTION](#) statements a routine can be defined which does return a value. The return value must be explicitly stated with the statement RETURN.

Variables which are used and declared within a SUB or FUNCTION by default have a global scope, meaning that they are seen by the main program and other routines. With the [LOCAL](#) statement variables can be declared local to the FUNCTION or SUB.

Mathematics, variables

The standard operators for mathematics can be used, like '+' for addition, '-' for subtraction, '/' for division and '*' for multiplication. For the binary 'and', the '&' symbol must be used, and for the binary 'or' use the pipe symbol '|'. Binary shifts are possible with '>>' and '<<'. The C operators '+=', '-=' and the like are not valid in BaCon.

Variable names may be of any length but cannot start with a number or an underscore symbol.

Arrays

An array will never be declared implicitly by BaCon, so arrays must be declared explicitly. This can be done by using the keyword [GLOBAL](#) for arrays which should be globally visible, or [LOCAL](#) for local array variables.

Arrays must be declared in the C syntax, using square brackets for each dimension. For example, a local string array must be declared like this: 'LOCAL array\$[5]'. Two-dimensional arrays are written like 'array[5][5]', three-dimensional arrays like 'array[5][5][5]' and so on.

By default, if an array is declared as 'array[5]', then it means that the array elements range from 0 to 4. Element 5 is not part of the array. This behavior can be changed using the [OPTION BASE](#) statement. If OPTION BASE is set to 1, the mentioned 'array[5]' will have a range from 1 to 5.

Arrays must be declared with fixed dimensions, meaning that it is not possible to determine the dimensions of an array using variables or functions, so during program runtime. The reason for this is that the C compiler needs to know the array dimensions during compile time. Therefore the dimensions of an array must be defined with fixed numbers or with [CONST](#) definitions.

In BaCon, numeric arrays can have all dimensions, but string arrays cannot have more than one dimension.

Associative arrays

Declaration

An associative array is an array of which the index is determined by a string, instead of a number. Associative arrays use round brackets '(...)' instead of the square brackets '[...]' used by normal arrays.

An associative array can use any kind of string for the index, and it can have an unlimited amount of elements. The declaration of associative arrays therefore never mentions the range. An associative array always has one dimension. Also note that the [OPTION BASE](#) statement has no impact.

To declare an associative array, the following syntax applies: [DECLARE](#) info ASSOC int. This declares an array containing integer values. To assign a value, using a random string "abcd" as

example: `info("abcd") = 1`. Similarly an associative array containing other types can be declared, for example strings: `DECLARE txt$ ASSOC STRING`. An associative array cannot be declared using the [LOCAL](#) keyword.

For the index, it is also possible to use the [STR\\$](#) function to convert numbers or numerical variables to strings: `PRINT txt$(STR$(123))`.

Relations, lookups

In BaCon it is possible to setup relations between associative arrays of the same type. This may be convenient when multiple arrays with the same index need to be set at once. To setup a relation the [RELATE](#) keyword can be used, e.g.: `RELATE assoc TO other`. Now for each index in the array 'assoc', the same index in the array 'other' is set.

Next to this the actual elements in an associative array can be looked up using the [LOOKUP](#) statement. This statement returns an array containing all indexes. The size of the array is dynamically declared as it depends on the amount of available elements.

Basic logic programming

With the current associative array commands it is possible to perform basic logic programming. Consider the following Logic program which can be executed with any Prolog implementation:

```
mortal(X) :- human(X).
```

```
human(socrates).
human(sappho).
human(august).
```

```
mortals_are:
    write('Mortals are:'),
    mortal(X),
    write(X),
    fail.
```

The following BaCon program does the same thing:

```
DECLARE human, mortal ASSOC int
RELATE human TO mortal

human("socrates") = TRUE
human("sappho") = TRUE
human("august") = TRUE

PRINT "Mortals are:"
LOOKUP mortal TO member$ SIZE amount
FOR x = 0 TO amount - 1
    PRINT member$(x)
NEXT
```

Strings by value or by reference

Strings can be stored *by value* or *by reference*. By value means that a copy of the original string is stored in a variable. This happens automatically when when a string variable name ends with the '\$' symbol.

Sometimes it may be necessary to refer to a string by reference. In such a case, simply declare a variable name as STRING but omit the '\$' at the end. Such a variable will point to the same memory location as the original string. The following examples should show the difference between by value and by reference.

When using string variables *by value*:

```
a$ = "I am here"
b$ = a$
a$ = "Hello world..."
PRINT a$, b$
```

This will print "Hello world...I am here". The variables point to their own memory area so they contain different strings. Now consider the following code:

```
a$ = "Hello world..."
LOCAL b TYPE STRING
b = a$
a$ = "Goodbye..."
PRINT a$, b FORMAT "%s%s\n"
```

This will print "Goodbye...Goodbye..." because the variable 'b' points to the same memory area as 'a\$'. (The optional FORMAT forces the variable 'b' to be printed as a string, otherwise BaCon assumes that the variable 'b' contains a value.)

Execute BaCon source program using a 'shebang'

In Unix, it is possible to execute a script directly from the commandline. The first line of the script must describe the interpreter. This is called a [shebang](#). A similar thing can be done with BaCon. If the first line of the BaCon program contains a shebang, the program will be compiled automatically. If the shebang also adds the '-b' option, the program will be compiled and also executed, as if it were a script. For example:

```
#!/dir/to/bacon -b
a$ = "Run from shebang"
PRINT a$
```

The first line points to the BaCon binary using the '-b' option. Also make sure to set the executable rights of the BaCon source program. Now the program can be executed like any other script. So if the program is called 'shebang.bac', then from the Unix commandline just run the following to compile and execute it:

```
./shebang.bac
```

Note that this way of executing a BaCon source program only can be performed with the binary version of BaCon.

Creating and linking to libraries created with BaCon

With Bacon, it is possible to create libraries. In the world of Unix these are known as *shared objects*. The following steps should be sufficient to create and link to BaCon libraries.

Step 1: create a library

The below program only contains a function, which accepts one argument and returns a value.

```
FUNCTION bla (NUMBER n)
  LOCAL i
  i = 5 * n
  RETURN i
END FUNCTION
```

In this example the program will be saved as 'libdemo.bac'. Note that the name *must* begin with the prefix 'lib'. This is a Unix convention. The linker will search for library names starting with these three letters.

Step 2: compile the library

The program must be compiled using the '-f' flag: `bacon -f libdemo.bac`

This will create a file called 'libdemo.so'.

Step 3: copy library to a system path

To use the library, it must be located in a place which is known to the linker. There are more ways to achieve this. For sake of simplicity, in this example the library will be copied to a system location. It is common usage to copy additional libraries to '/usr/local/lib': `sudo cp libdemo.so /usr/local/lib`

Step 4: update linker cache

The linker now must become aware that there is a new library. Update the linker cache with the following command: `sudo ldconfig`

Step 5: demonstration program

The following program uses the function from the new library:

```
PROTO bla
x = 5
result = bla(x)
PRINT result
```

This program first declares the function 'bla' as prototype, so the BaCon parser will not choke on this external function. Then the external function is invoked and the result is printed on the screen.

Step 6: compile and link

Now the program must be compiled with reference to the library created before. This can be done as follows: `./bacon -l "-ldemo" program.bac`

With the Unix command 'ldd' it will be visible that the resulting binary indeed has a dependency with the new library!

When executed, the result of this program should show 25.

Statements and functions

ABS

ABS(x)

Type: function

Returns the absolute value of x.

ADDRESS

ADDRESS(x)

Type: function

Returns the memory address of a variable or function. The ADDRESS function can be used when passing pointers to imported C functions (see [IMPORT](#)).

ALARM

ALARM <sub>, <time>

Type: statement

Sets a [SUB](#) to be executed in <time> seconds. This will interrupt any action the BaCon currently is performing; an alarm always has priority. Example:

```
SUB dinner
    PRINT "Dinner time!"
END SUB
ALARM dinner, 5
```

ALIAS

ALIAS <function> **TO** <alias>

Type: statement

Defines an alias to an existing function or an imported function. Aliases cannot be created for statements or operators. Example:

```
ALIAS "DEC" TO "ConvertToDecimal"
PRINT ConvertToDecimal("AB1E")
```

AND

x **AND** y

Type: operator

Performs a logical 'and' between x and y. For the binary 'and', use the '&' symbol.

ARGUMENT\$

ARGUMENT\$

Type: variable

Reserved variable containing name of the program and the arguments to the program. These are all separated by spaces.

ASC

ASC(char)

Type: function

Calculates the ASCII value of char (opposite of [CHR\\$](#)). Example:

```
PRINT ASC("x")
```

CALL

CALL <sub name> [**TO** <var>]

Type: statement

Calls a subroutine if the sub is defined at the end of the program. With the optional TO also a function can be invoked which stores the result value in <var>. Example:

```
CALL fh2cel(72) TO celsius
PRINT celsius
```

CATCH

CATCH GOTO <label> | **RESET**

Type: statement

Sets the error function where the program should jump to if error checking is enabled with [TRAP](#). For an example, see the [RESUME](#) statement. Using the RESET argument restores the BaCon default error messages.

CHANGEDIR

CHANGEDIR <directory>

Type: statement

Changes the current working directory. Example:

```
CHANGEDIR "/tmp/mydir"
```

CHOP\$

CHOPS(x)

Type: function

Returns a string where on both sides <CR>, <NL>, <TAB> and <SPACE> have been removed.

CHR\$

CHR(x)

Type: function

Returns the character belonging to ASCII number x. This function does the opposite of [ASC](#). The value for x must lay between 0 and 255.

CLEAR

CLEAR

Type: statement

Clears the terminal screen. To be used with ANSI compliant terminals.

CLOSE

CLOSE FILE|DIRECTORY|NETWORK|MEMORY <handle>

Type: statement

Close file, directory or network identified by handle. Example:

```
CLOSE FILE myfile
```

COLOR

COLOR <BG|FG> **TO** <BLACK|RED|GREEN|YELLOW|BLUE|MAGENTA|CYAN|WHITE>

COLOR <NORMAL|INTENSE|INVERSE|RESET>

Type: statement

Sets coloring for the output of characters in a terminal screen. For FG, the foreground color is set. With BG, the background color is set. This only works with ANSI compliant terminals. Example:

```
COLOR FG TO GREEN
PRINT "This is green!"
COLOR RESET
```

COLUMNS

COLUMNS

Type: function

Returns the amount of columns in the current ANSI compliant terminal.

CONCAT\$

CONCAT\$(x\$, y\$, ...)

Type: function

Returns the concatenation of x\$, y\$, ... The CONCAT function can accept an unlimited amount of arguments. Example:

```
PRINT CONCAT$("Help this is ", name$, " carrying a strange ",
thing$)
```

CONST

CONST <var> = <value> | <expr>

Type: statement

Assigns a value a to a label which cannot be changed during execution of the program. Consts are globally visible from the point where they are defined. Example:

```
CONST WinSize = 100
CONST Screen = WinSize * 10 + 5
```

COPY

COPY <file> TO <newfile>

Type: statement

Copies a file to a new file. Example:

```
COPY "file.txt" TO "/tmp/new.txt"
```

COS

COS(x)

Type: function

Returns the calculated COSINUS of x.

CURDIR\$

CURDIR\$

Type: function

Returns the full path of the current working directory.

CURSOR

CURSOR <ON|OFF>

Type: statement

Shows or hides the cursor in the current ANSI compliant terminal.

DATA

DATA <x, y, z....>

Type: statement

Defines float, integer or string data. The DATA statement always is globally visible. In a DATA statement there can only be one type of data. The data can be read with the [READ](#) statement. If more data is read than available, then in case of numeric data a '0' will be retrieved, and in case of string data an empty string. Example:

```
DATA 1, 2, 3, 4, 5, 6
```

```
DATA 0.5, 0.7, 0.11, 0.15
```

```
DATA "one", "two", "three", "four"
```

DAY

DAY(x)

Type: function

Returns the day of the month (1-31) where x is amount of seconds since January 1, 1970. Example:
PRINT DAY(NOW)

DEC

DEC(x)

Type: function

Calculates the decimal value of x, where x should be passed as a string. Example:
PRINT DEC("AB1E")

DECLARE

DECLARE <var>[,var2,var3,...] TYPE|ASSOC <c-type>

Type: statement

Explicitly declares a variable to a C-type. The ASSOC keyword is used to declare associative arrays. This is always a global declaration. Use [LOCAL](#) for local declarations. This function is available for compatibility reasons. It does the same thing as the [GLOBAL](#) statement.

Optionally, within a [SUB](#) or [FUNCTION](#) it is possible to use DECLARE in combination with [RECORD](#) to define a record variable which is visible globally.

DECR

DECR <x>[, y]

Type: statement

Decreases variable <x> with 1. Optionally, the variable <x> can be decreased by <y>. Example:

```
x = 10
```

```
DECR x
```

```
PRINT x
```

```
DECR x, 3
```

```
PRINT x
```

DEF FN

DEF FN <label> [(args)] = <value> | <expr>

Type: statement

Assigns a value or expression to a label. Example:

```
DEF FN func(x) = 3 * x
```

```
PRINT func(12)
```

DELETE

DELETE <FILE|DIRECTORY> <name>

Type: statement

Deletes a file or directory. If an error occurs then this can be retrieved with the [CATCH](#) statement.

Example:

```
DELETE FILE "/tmp/data.txt"
```

END

END [value]

Type: statement

Exits a program. Optionally, a value can be provided which the program can return to the shell.

ENDFILE

ENDFILE(filehandle)

Type: function

Function to check if EOF on a file opened with <handle> is reached. If the end of a file is reached, the value '1' is returned, else this function returns '0'. For an example, see the [OPEN](#) statement.

EQ

x **EQ** y

Type: operator

Verifies if x is equal to y. This is an alias for the C construct '=='. To improve readability it is also

possible to use IS instead. Example:

```
IF q EQ 5 THEN
    PRINT "q equals 5"
END IF
```

EQUAL

EQUAL(x\$, y\$)

Type: function

Returns 1 if x\$ and y\$ are equal, or 0 if x\$ and y\$ are not equal. This is the only way in BaCon to compare two strings. Use [OPTION COMPARE](#) to establish case insensitive conversion. Example:

```
IF EQUAL(a$, "Hello") THEN
    PRINT "world"
END IF
```

ERR\$

ERR\$(x)

Type: function

Returns the runtime error as a human readable string, identified by x. Example:

```
PRINT ERR$(ERROR)
```

ERROR

ERROR

Type: variable

This is a reserved variable, which contains the last [error number](#). This variable may be reset during runtime.

EVEN

EVEN(x)

Type: function

Returns 1 if x is even, else returns 0.

EXEC\$

EXEC\$(command\$)

Type: function

Executes an operating system command and returns the result to the BaCon program. See [SYSTEM](#) to plainly execute a system command. Example:

```
result$ = EXEC$("ls -l")
```

FALSE

FALSE

Type: variable
Represents and returns the value of '0'.

FILEEXISTS

FILEEXISTS(filename)

Type: function

Verifies if <filename> exists. If the file exists, this function returns 1, else it returns 0.

FILELEN

FILELEN(filename)

Type: function

Returns the size of a file identified by <filename>. If an error occurs this function returns '-1'. The [ERRS](#) statement can be used to find out the error if [TRAP](#) is set to LOCAL. Example:

```
length = FILELEN("/etc/passwd")
```

FILETYPE

FILETYPE(filename)

Type: function

Returns the type of a file identified by <filename>. If an error occurs this function returns '0'. The [ERRS](#) statement can be used to find out which error if [TRAP](#) is set to LOCAL. The following values may be returned:

Value	Meaning
1	Regular file
2	Directory
3	Character device
4	Block device
5	Named pipe (FIFO)
6	Symbolic link
7	Socket

FILL\$

FILL\$(x, y)

Type: function

Returns an x amount of ASCII character y. The value for y must lay between 0 and 255. Example printing 10 times the character '@':

```
PRINT FILL$(10, 64)
```

FLOOR

FLOOR(x)

Type: function

Returns the rounded down value of x.

FOR

FOR var = x **TO** y [**STEP** z]

<body>

[**BREAK**]

NEXT [var]

Type: statement

With FOR/NEXT a body of statements can be repeated a fixed amount of times. The variable will be increased with 1 unless a STEP is specified. Example:

```
FOR x = 1 TO 10 STEP 0.5
```

```
    PRINT x
```

```
NEXT
```

FREE

FREE x

Type: statement

Releases claimed memory (see also [MEMORY](#)).

FUNCTION

FUNCTION <name> ()(STRING s, NUMBER i, FLOATING f)

<body>

RETURN <x>

ENDFUNCTION | **END FUNCTION**

Type: statement

Defines a function. The variables within a function are visible globally, unless declared with the [LOCAL](#) statement. A FUNCTION always returns a value, this should explicitly be specified with the RETURN statement. Instead of the Bacon types STRING, NUMBER and FLOATING for the incoming arguments, also regular C-types can be used. Example:

```
FUNCTION fh2cel (NUMBER fahrenheit)
```

```
    LOCAL celsius
```

```
    celsius = fahrenheit*9/5 + 32
```

```
    RETURN celsius
```

```
END FUNCTION
```

GETBYTE

GETBYTE <memory> **FROM** <filehandle> [**SIZE** x]

Type: statement

Retrieves binary data into a memory area from a file identified by handle, with optional size of x bytes (default size = 1 byte). This command is useful in case binary data must be fetched. Example:

```
OPEN prog$ FOR READING AS myfile
    txt = MEMORY(100)
    GETBYTE txt FROM myfile SIZE 100
CLOSE FILE myfile
```

GETENVIRON\$

GETENVIRON\$(var\$)

Type: function

Returns the value of the environment variable 'var\$'. If the environment variable does not exist, this function returns an empty string. See [SETENVIRON](#) to set an environment variable.

GETFILE

GETFILE <var> FROM <dirhandle>

Type: statement

Reads a file from an opened directory. Subsequent reads return the files in the directory. If there are no more files then an empty string is returned. Refer to the [OPEN](#) statement for an example on usage.

GETKEY

GETKEY

Type: function

Returns a key from the keyboard without waiting for <RETURN>-key. Example:

```
PRINT "Press <escape> to exit now..."
```

```
key = GETKEY
```

```
IF key EQ 27 THEN
```

```
    END
```

```
END IF
```

GETLINE

GETLINE <var> FROM <handle>

Type: statement

Reads a line of data from a memory area identified by <handle> into a variable. The memory area can be opened in streaming mode using the the [OPEN](#) statement. A line of text is read until the next newline character. Example:

```
GETLINE txt$ FROM mymemory
```

See also [PUTLINE](#) to store lines of text into memory areas.

GETX / GETY

GETX

GETY

Type: function

Returns the current x and y position of the cursor. An ANSI compliant terminal is required. See [GOTOXY](#) to set the cursor position.

GLOBAL

GLOBAL <var>[,var2,var3,...] **TYPE|ASSOC** <c-type>

Type: statement

Defines a global variable <var> with C type <type>. The ASSOC keyword is used to declare associative arrays. In most cases BaCon tries to declare variables implicitly, so in most cases there is no need to use this statement. Variables declared with the GLOBAL keyword are visible in each part of the program. See also the [LOCAL](#) keyword for local declarations. Example:

```
GLOBAL x TYPE float
```

GOTO

GOTO <label>

Type: statement

Jumps to a label defined elsewhere in the program. See also the [LABEL](#) statement.

GOTOXY

GOTOXY x, y

Type: statement

Puts cursor to position x,y where 0,0 is the upper left of the terminal screen. An ANSI compliant terminal is required. Example:

```
CLEAR
FOR x = 5 TO 10
    GOTOXY x, x
    PRINT "Hello world"
NEXT
GOTOXY 1, 12
```

HEX\$

HEX\$(x)

Type: function

Calculates the hexadecimal value of x. Returns a string with the result.

HOUR

HOUR(x)

Type: function

Returns the hour (0-23) where x is the amount of seconds since January 1, 1970.

IF

```
IF <expression> THEN  
  <body>
```

```
[ELIF  
  <body>
```

```
[ELSE  
  [body]
```

```
ENDIF | END IF
```

Type: statement

Execute <body> if <expression> is true. If <expression> is not true then run the optional ELSE body. Multiple IF's can be written with ELIF. The IF construction should end with ENDFIF or END IF. Example:

```
a = 0
```

```
IF a > 10 THEN  
  PRINT "This is strange:"  
  PRINT "a is bigger than 10"
```

```
ELSE  
  PRINT "a is smaller than 10"
```

```
END IF
```

If only one function or statement has to be executed, then the if-statement also can be used without a body. For example:

```
IF age > 18 THEN PRINT "You are an adult"
```

```
ELSE INPUT "Your age: ", age
```

IMPORT

```
IMPORT <function[(type arg1, type arg2, ...)]> FROM <library> TYPE <type> [ALIAS word]
```

Type: statement

Import a function from a C library defining the type of returnvalue. Optionally, the type of arguments can be specified. Also optionally it is possible to define an alias under which the imported function will be known to BaCon. Examples:

```
IMPORT "ioctl" FROM "libc.so" TYPE int
```

```
IMPORT "gdk_draw_line(long, long, int, int, int, int)" FROM  
"libgdk-x11-2.0.so" TYPE void
```

```
IMPORT "atan(double)" FROM "libm.so" TYPE double ALIAS  
"arctangens"
```

INCLUDE

```
INCLUDE <filename>
```

Type: statement

Adds a external BaCon file to current program. Nested includes are not supported. Example:

```
INCLUDE "beep.bac"
```

INCR

```
INCR <x>[, y]
```

Type: statement

Increases variable <x> with 1. Optionally, the variable <x> may be increased by <y>.

INPUT

INPUT [text[, ... ,]<variable[\$]>

Type: statement

Gets input from the user. If the variable ends with a '\$' then the input is considered to be a string. Otherwise it will be treated as numeric. Example:

```
INPUT a$
```

```
PRINT "You entered the following: ", a$
```

The input-statement also can print text. The input variable always must be present at the end of the line. Example:

```
INPUT "What is your age? ", age
```

```
PRINT "You probably were born in ", YEAR(NOW) - age
```

INSTR

INSTR(haystack\$, needle\$ [,z])

Type: function

Returns the position where needle\$ begins in haystack\$, optionally starting at position z. If not found then this function returns the value '0'.

INSTREV

INSTREV(haystack\$, needle\$ [,z])

Type: function

Returns the position where needle\$ begins in haystack\$, but start searching from the end of haystack\$, optionally at position z also counting from the end. If not found then this function returns the value '0'.

ISFALSE

ISFALSE(x)

Type: function

Verifies if x is equal to 0.

ISTRUE

ISTRUE(x)

Type: function

Verifies if x is not equal to 0.

LABEL

LABEL <label>

Type: statement

Defines a label which can be jumped to with a [GOTO](#) statement. A label may not contain spaces.

LCASE\$

LCASE\$(x\$)

Type: function

Converts x\$ to lowercase characters and returns the result. Example:

```
PRINT LCASE$("ThIs Is All LoWeRcAsE")
```

LEFT\$

LEFT\$(x\$, y)

Type: function

Returns y characters from the left of x\$.

LEN

LEN(x\$)

Type: function

Returns the length of x\$.

LET

LET <var> = <value> | <expr>

Type: statement

Assigns a value or result from an expression to a variable. The LET statement may be omitted.

Example:

```
LET a = 10
```

LOCAL

LOCAL <var>[,var2,var3,...] [TYPE <c-type>]

Type: statement

This statement only has sense within functions, subroutines or records. It defines a local variable <var> with C type <type> which will not be visible for other functions, subroutines or records, nor for the main program. If the TYPE keyword is omitted then variables are assumed to be of 'long' type. If TYPE is omitted and the variable name ends with a '\$' then the variable will be a string.

Example:

```
LOCAL x, y TYPE int
```

```
LOCAL q$
```

LOOKUP

LOOKUP <assoc> TO <array> SIZE <variable>

Type: statement

Retrieves all index names created in an associative array. The results are stored in <array>. As it

sometimes is unknown how many elements this resulting array will contain, the array should not be declared explicitly. Instead, LOOKUP will declare the result array dynamically.

If LOOKUP is being used in a function or sub, then <array> will have a local scope. Else <array> will be visible globally, and can be accessed within all functions and subs.

The total amount of elements created in this array is stored in <variable>. This variable can be declared explicitly using [LOCAL](#) or [GLOBAL](#). Example:

```
LOOKUP mortal TO men$ SIZE amount
FOR x = 0 TO amount - 1
    PRINT men$[x]
NEXT
```

MAKEDIR

MAKEDIR <directory>

Type: statement

Creates an empty directory. Parent directories are created implicitly. Errors like write permissions, disk quota issues and so on can be retrieved with [CATCH](#). Example:

```
MAKEDIR "/tmp/mydir/is/here"
```

MAXRANDOM

MAXRANDOM

Type: variable

Reserved variable which contains the maximum value [RND](#) can generate. The actual value may vary on different operating systems.

MEMORY

MEMORY(x)

Type: function

Claims memory of x byte size, returning the address where the memory block resides. Use [FREE](#) to release the memory again.

MEMREWIND

MEMREWIND <handle>

Type: statement

Returns to the beginning of a memory area opened with <handle>.

MEMTELL

MEMTELL(handle)

Type: function

Returns the current position in the memory area opened with <handle>.

MID\$

MID\$(x\$, y, [z])

Type: function

Returns z characters starting at position y in x\$. The parameter 'z' is optional. When omitted, this function returns everything from position 'y' until the end of the string.

MINUTE

MINUTE(x)

Type: function

Returns the minute (0-59) where x is amount of seconds since January 1, 1970.

MOD

MOD(x, y)

Type: function

Returns the modulo of x divided by y.

MONTH

MONTH(x)

Type: function

Returns the month (1-12) in a year, where x is the amount of seconds since January 1, 1970.

MONTH\$

MONTH\$(x)

Type: function

Returns the month of the year as string in the system's locale ("January", "February", etc), where x is the amount of seconds since January 1, 1970.

NE

x **NE** y

Type: operator

Checks if x and y are not equal. This is an alias for the C construct '!='. Instead, ISNOT can be used as well to improve code readability.

NL\$

NL\$

Type: variable

Represents the newline as a string.

NOT

NOT(x)

Type: function

Returns the negation of x.

NOW

NOW

Type: function

Returns the amount of seconds since January 1, 1970.

ODD

ODD(x)

Type: Function

Returns 1 if x is odd, else returns 0.

OPEN

OPEN <file|dir|address> **FOR READING|WRITING|APPENDING|READWRITE|**
DIRECTORY|NETWORK|SERVER|MEMORY AS <handle>

Type: statement

When used with READING, WRITING, APPENDING or READWRITE, this statement opens a file assigning a handle to it. The READING keyword opens a file for read-only, the WRITING for writing, APPENDING to append data and READWRITE opens a file both for reading and writing.

Example:

```
OPEN "data.txt" FOR READING AS myfile
WHILE NOT(ENDFILE(myfile)) DO
    READLN txt$ FROM myfile
    IF NOT(ENDFILE(myfile)) THEN
        PRINT txt$
    ENDIF
WEND
```

```
CLOSE FILE myfile
```

When used with DIRECTORY a directory is opened as a stream. Subsequent reads will return the files in the directory. Example:

```
OPEN "." FOR DIRECTORY AS mydir
REPEAT
```

```
    GETFILE myfile$ FROM mydir
    PRINT "File found: ", myfile$
```

```
UNTIL ISFALSE(LEN(myfile$))
CLOSE DIRECTORY mydir
```

When used with NETWORK a network address is opened as a stream.

```
OPEN "www.google.com:80" FOR NETWORK AS mynet
SEND "GET / HTTP/1.1\r\nHost: www.google.com\r\n\r\n" TO mynet
REPEAT
```

```
    RECEIVE dat$ FROM mynet
    total$ = CONCAT$(total$, dat$)
```

```
UNTIL ISFALSE(WAIT(mynet, 500))
PRINT total$
CLOSE NETWORK mynet
```

When used with SERVER the program starts as a server to accept incoming TCP connections.

```
OPEN "localhost:51000" FOR SERVER AS myserver
WHILE NOT(EQUAL(LEFT$(dat$, 4), "quit")) DO
    RECEIVE dat$ FROM myserver
    PRINT "Found: ", dat$
WEND
CLOSE SERVER myserver
```

When used with MEMORY a memory area can be used in streaming mode.

```
data = MEMORY(500)
OPEN data FOR MEMORY AS mem
PUTLINE "Hello cruel world" TO mem
MEMREWIND mem
GETLINE txt$ FROM mem
CLOSE MEMORY mem
PRINT txt$
```

OPTION

OPTION <BASE x> | <COMPARE x> | <SOCKET x>

Type: statement

Sets an option to define the behavior of the compiled BaCon program. It is recommended to use this statement in the beginning of the program, to avoid unexpected results.

- The BASE argument determines the lower bound of arrays. By default the lower bound is set to 0. Note that this setting also has impact on the array returned by the [SPLIT](#) statement.
- The COMPARE argument defines if string comparisons with [EQUAL](#) should be case sensitive (0) or not (1). The default is *case sensitive* comparison (0).
- The SOCKET argument defines the timeout for setting up a socket to an IP address. Default value is 5 seconds.

OR

x **OR** y

Type: operator

Performs a logical or between x and y. For the binary or, use the '|' symbol.

OS\$

OS\$

Type: function

Function which returns the name and machine of the current Operating System.

PEEK

PEEK(x)

Type: function

Returns a 1-byte value stored at memory address x.

PI

PI

Type: variable

Reserved variable containing the number for PI: 3.14159265.

POKE

POKE <x>, <y>

Type: statement

Stores a 1-byte value <y> at memory address <x>. Use [PEEK](#) to retrieve a value from a memory address. Example:

```
mem = MEMORY(500)
```

```
POKE mem, 32
```

POW

POW(x, y)

Type: function

Raise x to the power of y.

PRINT

PRINT [value] | [text] | [variable] | [expression] [**FORMAT** <format>] | [,] | [;]

Type: statement

Prints a numeric value, text, variable or result from expression. A semicolon at the end of the line prevents printing a newline. Example:

```
PRINT "This line does ";
```

```
PRINT "end here: ";
```

```
PRINT linenr + 2
```

Multiple arguments maybe used but they must be separated with a comma. Examples:

```
PRINT "This is operating system: ", OS$
```

```
PRINT "Sum of 1 and 2 is: ", 1 + 2
```

The **FORMAT** argument is optional and can be used to specify different types in the **PRINT** argument. The syntax of **FORMAT** is similar to the **printf** argument in C. Example:

```
PRINT "My age is ", 40, " years which is ", 10 + 30 FORMAT "%s%d%s  
%d\n"
```

PROTO

PROTO <function name>[,function name [, ...]] [**ALIAS** word]

Type: statement

Defines an external function so it is accepted by the BaCon parser. Multiple function names may be mentioned but these should be separated by a comma. Optionally, **PROTO** accepts an alias which can be used instead of the original function name. During compilation the BaCon program must

explicitly be linked with an external library to resolve the function name. Examples:

```
PROTO glClear, glClearColor, glEnable  
PROTO "glutSolidTeapot" ALIAS "TeaPot"
```

PULL

PULL x

Type: statement

Puts a value from the internal stack into variable <x>. The argument must be a variable. The stack will decrease to the next available value.

If the internal stack has reached its last value, subsequent PULL's will retrieve this last value. If no value has been pushed before, a PULL will deliver 0 for numeric values and an empty string for string values.

PUSH

PUSH <x>|<expression>

Type: statement

Pushes a value <x> or expression to the internal stack. There is no limit to the amount of values which can be put onto the stack other than the available memory. The principle of the stack is Last In, First Out.

See also [PULL](#) to get a value from the stack.

PUTBYTE

PUTBYTE <memory> **TO** <filehandle> [**SIZE** x]

Type: statement

Store binary data from a memory area to a file identified by handle with optional size of x bytes (default size = 1 byte). This is the inverse of [GETBYTE](#), refer to this command for an example.

PUTLINE

PUTLINE "text"|<var> **TO** <handle>

Type: statement

Write a line of data to a memory area identified by handle. The line will be terminated by a newline character. The memory area must be set in streaming mode first using [OPEN](#). Example:

```
PUTLINE "hello world" TO mymemory
```

RANDOM

RANDOM (x)

Type: function

This is a convenience function to generate a random integer number between 0 and x - 1. See also [RND](#) for more flexibility in creating random numbers. Example creating a random number between 1 and 100:

```
number = RANDOM(100) + 1
```

READ

READ <x[\$]>

Type: statement

Reads a value from a [DATA](#) block into variable <x>. Example:

```
LOCAL dat[8]
FOR i = 0 TO 7
    READ dat[i]
NEXT
DATA 10, 20, 30, 40, 50, 60, 70, 80
```

READLN

READLN <var> **FROM** <handle>

Type: statement

Reads a line of data from a file identified by handle into a variable. Example:

```
READLN txt$ FROM myfile
```

RECEIVE

RECEIVE <var> **FROM** <handle> [**SIZE** chunk]

Type: statement

Reads data from a network location identified by handle into a variable. Subsequent reads return more data until the network buffer is empty. The chunk size can be determined with the optional **SIZE** argument. Example:

```
OPEN "www.google.com:80" FOR NETWORK AS mynet
SEND "GET / HTTP/1.1\r\nHost: www.google.com\r\n\r\n" TO mynet
REPEAT
    RECEIVE dat$ FROM mynet
    total$ = CONCAT$(total$, dat$)
UNTIL ISFALSE(WAIT(mynet, 500))
CLOSE NETWORK mynet
```

RECORD

RECORD <var>

LOCAL <member1> **TYPE** <type>

LOCAL <member2> **TYPE** <type>

END RECORD

Type: statement

Defines a record <var> with members. If the record is defined in the mainprogram, it automatically will be globally visible. If the record is defined within a function, the record will have a local scope, meaning that it is only visible within that function. To declare a global record in a function, use the [DECLARE](#) or [GLOBAL](#) keyword.

The members of a record should be defined using the [LOCAL](#) statement and can be accessed with the 'var.member' notation. Also refer to [WITH](#) for assigning values to multiple members at the same time. Example:

```
RECORD var
```

```
LOCAL x
LOCAL y
END RECORD
var.x = 10
var.y = 20
PRINT var.x + var.y
```

REGEX

REGEX (txt\$, expr\$)

Type: function

Applies a [POSIX Extended Regular Expression](#) expr\$ to the string txt\$. If the expression matches, the value '1' is returned. Else this function returns '0'. Examples:

' Does the string match alphanumeric character

```
PRINT REGEX("Hello world", "[[:alnum:]]")
```

' Does the string *not* match a number

```
PRINT REGEX("Hello world", "[^0-9]")
```

' Does the string contain an a, l or z

```
PRINT REGEX("Hello world", "a|l|z")
```

RELATE

RELATE <assocA> **TO** <assocB>[, assocC, ...]

Type: statement

This statement creates a relation between associative arrays. Effectively this will result into duplication of settings; an index in array <assocA> also will be set in array <assocB>. A previous declaration of the associative arrays involved is required. Example:

```
DECLARE human, mortal ASSOC int
```

```
RELATE human TO mortal
```

```
human("socrates") = TRUE
```

```
PRINT mortal("socrates")
```

REM

REM [remark]

Type: statement

Adds a comment to your code. Any type of string may follow the REM statement. Instead of REM also the single quote symbol ' maybe used to insert comments in the code.

RENAME

RENAME <filename> **TO** <new filename>

Type: statement

Renames a file. If different paths are included the file is moved from one path to the other.

Example:

```
RENAME "tmp.txt" TO "real.txt"
```

REPEAT

REPEAT

<body>

[BREAK]

UNTIL <expr>

Type: statement

The REPEAT/UNTIL construction repeats a body of statements. The difference with [WHILE/WEND](#) is that the body will be executed at least once. The optional BREAK statement can be used to break out the loop. Example:

```
REPEAT
```

```
    C = GETKEY
```

```
UNTIL C EQ 27
```

REPLACE\$

REPLACE\$(haystack\$, needle\$, replacement\$)

Type: function

Substitutes a substring <needle\$> in <haystack\$> with <replacement\$> and returns the result. The replacement does not necessarily need to be of the same size as the substring. Example:

```
PRINT REPLACE$("Hello world", "l", "p")
```

```
PRINT REPLACE$("Some text", "me", "123")
```

RESIZE

RESIZE <x>, <y>

Type: statement

Resizes memory area starting at address <x> to an amount of <y> bytes. If the area enlarged, the original contents of the area remain intact.

RESTORE

RESTORE

Type: statement

Restores the internal DATA pointer(s) to the beginning of the first [DATA](#) statement.

RESUME

RESUME

Type: function

When an error is caught, this statement tries to continue after the statement where an error occurred. Example:

```
TRAP LOCAL
```

```
CATCH GOTO print_err
```

```
DELETE FILE "somefile.txt"
```

```
PRINT "Resumed..."
```

```
END
```

```
LABEL print_err
```

PRINT ERR\$(ERROR)
RESUME

REVERSE\$

REVERSE\$(x\$)

Type: function

Returns the reverse of x\$.

REWIND

REWIND <handle>

Type: statement

Returns to the beginning of a file opened with <handle>.

RIGHT\$

RIGHT\$(x\$, y)

Type: function

Returns y characters from the right of x\$.

RND

RND

Type: function

Returns a random number between 0 and the reserved variable [MAXRANDOM](#). The generation of random numbers can be seeded with the statement [SEED](#). Example:

```
SEED NOW
```

```
x = RND
```

ROUND

ROUND(x)

Type: function

Rounds x to the nearest integer number. For compatibility reasons, also the keyword INT may be used instead.

ROWS

ROWS

Type: function

Returns the amount of rows in the current ANSI compliant terminal.

SEARCH

SEARCH(handle, string)

Type: function

Searches for a <string> in file opened with <handle>. Returns the offset in the file where the first occurrence of <string> is located. Use [SEEK](#) to effectively put the filepointer at this position. If the string is not found, then the value '-1' is returned.

SECOND

SECOND(x)

Type: function

Returns the second (0-59) where x is the amount of seconds since January 1, 1970.

SEED

SEED x

Type: statement

Seeds the random number generator with some value. After that, subsequent usages of [RND](#) will return numbers in a random order. Note that seeding the random number generator with the same number also will result in the same sequence of random numbers.

By default, a BaCon program will automatically seed the random number generator with the current time as soon as it is executed, so it may not be needed to use this function explicitly. Example:

```
SEED NOW
```

SEEK

SEEK <handle> **OFFSET** <offset> [**WHENCE** **START**|**CURRENT**|**END**]

Type: statement

Puts the filepointer to new position at <offset>, optionally starting from <whence>.

SELECT

SELECT <variable> **CASE** <body>[;] [**DEFAULT** <body>] **END SELECT**

Type: statement

With this statement a variable can be examined on multiple values. Optionally, if none of the values match the SELECT statement may fall back to the DEFAULT clause. Example:

```
SELECT myvar
  CASE 1
    PRINT "Value is 1"
  CASE 5
    PRINT "Value is 5"
  CASE 2*3
    PRINT "Value is ", 2*3
  DEFAULT
    PRINT "Value not found"
END SELECT
```

Contrary to most implementations, in BaCon the CASE keyword also may refer to expressions and

variables. Also BaCon knows how to 'fall through' using a semicolon, in case multiple values lead to the same result:

```
SELECT st$
  CASE "Man"
    PRINT "It's male"
  CASE "Woman"
    PRINT "It's female"
  CASE "Child";
  CASE "Animal"
    PRINT "It's it"
  DEFAULT
    PRINT "Value not found"
END SELECT
```

SEND

SEND <var> **TO** <handle>

Type: statement

Sends data to a network location identified by handle. For an example, see the [RECEIVE](#) statement.

SETENVIRON

SETENVIRON var\$, value\$

Type: statement

Sets the environment variable 'var\$' to 'value\$'. If the environment variable already exists, this statement will overwrite a previous value. See [GETENVIRON\\$](#) to retrieve the value of an environment variable. Example:

```
SETENVIRON "LANG", "C"
```

SIN

SIN(x)

Type: function

Returns the calculated SINUS of x

SLEEP

SLEEP <x>

Type: statement

Sleeps <x> milliseconds (sleep 1000 is 1 second).

SORT

SORT <x> [**DOWN**]

Type: statement

Sorts the one-dimensional array <x> in ascending order. Only the basename of the array should be mentioned, not the dimension. The array may both be a numeric or a string array. Optionally the

keyword DOWN can be used to sort in descending order. Example:

```
GLOBAL a$[5] TYPE STRING
a$[0] = "Hello"
a$[1] = "my"
a$[2] = "good"
a$[4] = "friend"
SORT a$
```

SPC\$

SPC\$(x)

Type: function

Returns an x amount of spaces.

SPLIT

SPLIT <string> **BY** <sub> **TO** <array> **SIZE** <variable>

Type: statement

This statement can split a string into smaller pieces. The <sub> argument determines where the string is being splitted. The results are stored in <array>. As it cannot be known in advance how many elements this resulting array will contain, the array may not be declared before with [LOCAL](#) or [GLOBAL](#).

If SPLIT is being used in a function or sub, then <array> will have a local scope. Else <array> will be visible globally, and can be accessed within all functions and subs.

The total amount of elements created in this array is stored in <variable>. This variable can be declared explicitly using [LOCAL](#) or [GLOBAL](#). Example usage:

```
OPTION BASE 1
LOCAL dimension
SPLIT "one two three" BY " " TO array$ SIZE dimension
FOR i = 1 TO dimension
    PRINT array$[i]
NEXT
```

SQR

SQR(x)

Type: function

Calculates the square root from a number.

STOP

STOP

Type: statement

Halts the current program and returns to the Unix prompt. The program can be resumed by performing the Unix command 'fg', or by sending the CONT signal to its pid: kill -CONT <pid>.

STR\$

STR\$(x)

Type: function

Convert numeric value x to a string (opposite of [VAL](#)). Example:

```
PRINT STR$(123)
```

SUB

SUB <name>[(STRING s, NUMBER i, FLOATING f)]

<body>

ENDSUB | **END SUB**

Type: statement

Defines a subprocedure. A subprocedure never returns a value (use [FUNCTION](#) instead).

Variables used in a sub are visible globally, unless declared with [LOCAL](#). The incoming arguments are always local. Instead of the BaCon types STRING, NUMBER and FLOATING for the incoming arguments, also regular C-types also can be used. Example:

```
SUB add(NUMBER x, NUMBER y)
    LOCAL result
    PRINT "The sum of x and y is: ";
    result = x + y
    PRINT result
END SUB
```

SYSTEM

SYSTEM <command\$>

Type: statement

Executes an operating system command. Use [EXEC\\$](#) to catch the result of an operating system command. Example:

```
SYSTEM "ls -l"
```

TAB\$

TAB\$(x)

Type: function

Returns an x amount of tabs.

TAN

TAN(x)

Type: function

Returns the calculated tangens of x.

TELL

TELL(handle)

Type: function
Returns current position in file opened with <handle>.

TIMEVALUE

TIMEVALUE(a,b,c,d,e,f)

Type: function

Returns the amount of seconds since January 1 1970, from year (a), month (b), day (c), hour (d), minute (e), and seconds (f). Example:

```
PRINT TIMEVALUE(2009, 11, 29, 12, 0, 0)
```

TRACE

TRACE <ON|OFF>

Type: statement

Starts trace mode. The program will wait for a key to continue. After each keypress, the next line of source code is printed to the screen, and then executed. Pressing the ESCAPE key will exit the program.

TRAP

TRAP <LOCAL|SYSTEM>

Type: statement

Sets the runtime error trapping. By default, trapping is performed by the operating system. This means that if an error occurs, a signal will be caught by the program and a generic error message is displayed on the prompt. The program will then exit gracefully. If trapping is put to LOCAL, BaCon tries to examine statements and functions where possible, and will display an error message based on the operating system internals, indicating which statement or function causes a problem. Optionally, when a [CATCH](#) is set, BaCon can jump to a [LABEL](#) instead, where a self-defined error function can be executed, and from where a [RESUME](#) is possible. Note that the [CATCH](#) statement does not work with functions.

The setting LOCAL decreases the performance of the program, because additional runtime checks are carried out when the program is executed.

TRUE

TRUE

Type: variable

Represents and returns the value of '1'. This is the opposite of the [FALSE](#) variable.

UCASE\$

UCASES(x\$)

Type: function

Converts x\$ to uppercase characters and returns the result. See [LCASE\\$](#) to do the opposite.

USEC

USEC

<body>

ENDUSEC | END USEC

Type: statement

Defines a body with C code. This code is passed unmodified to the C compiler. Example:

USEC

```
char *str;
str = strdup("Hello");
printf("%s\n", str);
```

END USEC

VAL

VAL(x\$)

Type: function

Returns the actual value of x\$. This is the opposite of [STR\\$](#). Example:

```
nr$ = "456"
```

```
q = VAL(nr$)
```

VERSION\$

VERSION\$

Type: variable

Reserved variable which contains the BaCon version number.

WAIT

WAIT(handle, milliseconds)

Type: function

Suspends the program for a maximum of <milliseconds> until data becomes available on <handle>.

This is especially useful in network programs where a [RECEIVE](#) will block if there is no data available. The WAIT function checks the handle and if there is data in the queue, it returns with value '1'. If there is no data then it waits for at most <milliseconds> before it returns. If there is no data available, WAIT returns '0'. Refer to the [RECEIVE](#) statement for an example.

WEEK

WEEK(x)

Type: function

Returns the week number (1-53) in a year, where x is the amount of seconds since January 1, 1970.

Example:

```
PRINT WEEK(NOW)
```

WEEKDAYS\$

WEEKDAYS\$(x)

Type: function

Returns the day of the week as a string in the system's locale ("Monday", "Tuesday", etc), where x is the amount of seconds since January 1, 1970.

WHILE

WHILE <expr> [**DO**]

<body>

[**BREAK**]

WEND

Type: statement

The WHILE/WEND is used to repeat a body of statements and functions. The DO keyword is optional. The optional BREAK statement can be used to break out the loop. Example:

```
LET a = 5
```

```
WHILE a > 0 DO
```

```
    PRINT a
```

```
    a = a - 1
```

```
WEND
```

WITH

WITH <var>

.<var> = <value>

.<var> = <value>

....

END WITH

Type: statement

Assign values to individual members of a [RECORD](#). For example:

```
WITH myrecord
```

```
    .name$ = "Peter"
```

```
    .age = 40
```

```
    .street = Westlands 1
```

```
    .city = The Hague
```

```
END WITH
```

WRITELN

WRITELN "text"|<var> **TO** <handle>

Type: statement

Write a line of data to a file identified by handle. Example:

```
WRITELN "hello world" TO myfile
```

YEAR

YEAR(x)

Type: function

Returns the year where x is amount of seconds since January 1, 1970. Example:

`PRINT YEAR(NOW)`

Appendix A: Runtime error codes

Code	Meaning
0	Success
1	Trying to access illegal memory
2	Error opening file
3	Could not open library
4	Symbol not found in library
5	Wrong hexvalue
6	Unable to claim memory
7	Unable to delete file
8	Could not open directory
9	Unable to rename file
10	NETWORK argument should contain colon with port number
11	Could not resolve hostname
12	Socket error
13	Unable to open address
14	Error reading from socket
15	Error sending to socket
16	Error checking socket
17	Unable to bind the specified socket address
18	Unable to listen to socket address
19	Cannot accept incoming connection
20	Unable to remove directory
21	Unable to create directory
22	Unable to change to directory
23	GETENVIRON argument does not exist as environment variable

24	Unable to stat file
25	Search contains illegal string
26	Cannot return OS name
27	Illegal regex expression
28	Unable to create bidirectional pipes
29	Unable to fork process
30	Cannot read from pipe

Created with OpenOffice 3.2.

[Back to top of document](#)