BaCon BASIC converter documentation - version 2.4

Contents

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. 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
```

Another possibility is to point to the URL of a BaCon program hosted by a website. The program will then be downloaded automatically, after which it is converted:

```
./bacon.bash http://www.basic-converter.org/fetch.bac
```

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
- -x: extract gettext strings from generated c sources
- -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)
- -w: store commandline settings in a configurationfile. This file will be used in subsequent invocations of BaCon (not applicable for the GUI version)
- -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 , optimize and strip: ./bacon -o -O2 -o -s 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 current platform: ./bacon -o -m32 -o -mtune=native yourprogram.bac
- Convert and compile program linking to a particular library: ./bacon -l somelib yourprogram.bac
- Convert and compile program including an additional C header file: ./bacon -i header.h 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 of the line. The LET statement may be omitted, so a line may contain an assignment. Expressions are not converted but are passed unchanged to the C compiler (*lazy conversion*).
BaCon does not need line numbers. More statements per line are accepted. These should be separated by the colon symbol ':'.
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. Functions always return a value or string, contrary to subs. Functions created in the BaCon program can be invoked standalone, meaning that they do not need to appear in an assignment.
A variable will be declared implicitly when the variable is used in an assignment or in a statement which assigns a value to a variable. Implicitly declared variables always have a global scope, meaning that they are visible to all functions and routines in the whole program. Variables which are used and declared within a SUB or FUNCTION also by default have a global scope. With the LOCAL statement variables can be declared local to the FUNCTION or SUB.
If a variable name ends with the '$' symbol, a string variable is assumed. Otherwise the variable 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 FUNCTON/ENDFUNCTION statements a routine can be defined which does return a value. The return value must be explicitly stated with the statement RETURN.

Mathematics, variables

The standard C operators for mathematics can be used, like '+' for addition, '-' for substraction, '/' 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 '<<'.

C operator	Meaning	C Operator	Meaning
+	Addition	|	Inclusive or
-	Substraction	^	Exclusive or
*	Multiplication	>>	Binary shift right

/	Division	<<	Binary shift left
&	Binary and	+=, -=, *=, /=	Invalid in BaCon

The C operators '+=', '-=' and the like are not valid in BaCon. Use INCR or DECR instead.
Variable names may be of any length but may not start with a number or an underscore symbol.

Equations

Equations are used in statements like IF...THEN, WHILE...WEND, and REPEAT...UNTIL. In BaCon the following symbols for equations can be used:

Symbol	Meaning	Type
=, ==	Equal to	String, numeric
!=, <>	Not equal to	String, numeric
>, GT	Greater than	Numeric
<, LT	Less than	Numeric
EQ, IS	Equal to	Numeric
>=, GE	Greater or equal	Numeric
<=, LE	Less or equal	Numeric
NE, ISNOT	Not equal to	Numeric
EQUAL()	Equal to	String

Indexed arrays

Declaration of static 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 or DECLARE 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.
In BaCon, static numeric arrays can have all dimensions, but static string arrays cannot have more than one dimension.

Declaration of dynamic arrays

Also dynamic arrays must be declared explicitly. To declare a dynamic array, the statements GLOBAL or LOCAL must be used together with the ARRAY keyword, which determines the amount of elements. For example, to declare a dynamic array of 5 integer elements: 'LOCAL array TYPE int ARRAY 5'.

The difference with a static array is that the size of a dynamic array can declared using variables, and that they can redimension their size during runtime. The latter can be achieved with the REDIM statement.
As with static numeric arrays, also dynamic numeric arrays can have all dimensions, and dynamic string arrays cannot have more than one dimension. The syntax to refer to elements in a dynamic array is the same as the syntax for elements in a static array.

Dimensions

Static 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. Also, the size of a static array cannot be changed afterwards.
Dynamic arrays however can be declared with variable dimensions, meaning that the size of an array also can be contained in a variable. Furthermore, the size of a one dimensional dynamic array can be changed afterwards with the REDIM statement.
By default, if an array is declared with 5 elements, 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, an array declared with 5 elements will have a range from 1 to 5.

Passing arrays to functions or subs

In BaCon it is possible to pass one-dimensional arrays to a function or sub. The caller should simply use the basename of the array (so without mentioning the dimension of the array).
When the function or sub argument mentions the dimension, a local copy of the array is created.

```
CONST dim = 2
DECLARE series[dim] TYPE NUMBER
SUB demo(NUMBER array[dim])
    array[0] = 987
    array[1] = 654
END SUB
series[0] = 123
series[1] = 456
demo(series)
FOR x = 0 TO dim - 1
    PRINT series[x]
NEXT
```

This will print the values originally assigned. The sub does not change the original assignments.
When the function or sub argument does not mention the dimension, but only uses square brackets, the array is passed by reference.

```
CONST dim = 2
DECLARE series[dim] TYPE NUMBER
SUB demo(NUMBER array[])
    array[0] = 987
    array[1] = 654
END SUB
series[0] = 123
series[1] = 456
demo(series)
FOR x = 0 TO dim - 1
    PRINT series[x]
NEXT
```

This will modify the original array and prints the values assigned in the sub.

Returning arrays from functions

In BaCon it is also possible to return a one dimensional array from a function. This can only be done with dynamic arrays, as the static arrays always use the stack memory assigned to a function. This means, that when a function is finished, also the memory for that function is destroyed, together with the variables and static arrays in that function.
The syntax to return a one dimensional dynamic array involves two steps: the declaration of the array must contain the STATIC keyword, and the RETURN argument should only contain the basename of the array without mentioning the dimensions. For example:

```
FUNCTION demo
    LOCAL array TYPE int ARRAY 10 STATIC
    FOR x = 0 TO 9
        array[x] = x
    NEXT
    RETURN array
END FUNCTION
DECLARE my_array TYPE int ARRAY 10
my_array = demo()
```

This example will create a dynamic array and assign some initial values, after which it is returned from the function. The target 'my_array' now will contain the values assigned in the function.
The statements SPLIT and LOOKUP also accept the STATIC keyword, which allows the implicitly created dynamic array containing results to be returned from a function.
Note that when returning arrays, the assigned array should have the same dimensions in order to prevent memory errors.

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 can have any amount of dimension. 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, keys

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 a dynamically created array containing all indexes. The size of the resulting array is dynamically declared as it depends on the amount of available elements.
To find out if a key already was defined in the associative array, the function ISKEY can be used. This function needs the array name and the string containing the index name, and will return either TRUE or FALSE, depending on whether the index is defined (TRUE) or not (FALSE).
Deleting individual associative array members can be done by using the FREE statement. This will leave the associative array insertion order intact.

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 individual memory areas 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 demo 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.

Creating internationalization files

It is possible to create internationalized strings for a BaCon program. In order to do so, OPTION INTERNATIONAL should be enabled in the beginning of the program. After this, make sure that each translatable string is surrounded by the INTL$ or NNTL$ function.
Now start BaCon and use the '-x' option. This will generate a template for the catalog file, provided that the 'xgettext' utility is available on your platform. The generated template by default has the same name as your BaCon program, but with a '.pot' extension.
Then proceed with the template file and fill in the needed translations, create the PO file as usual and copy the binary formatted catalog to the base directory of the catalog files (default: "/usr/share/locale").
The default textdomain and base directory can be changed with the TEXTDOMAIN statement.
Below a complete sequence of steps creating internationalization files. Make sure the GNU gettext utilities are installed.

Step 1: create program

The following simple program should be translated:

```
OPTION INTERNATIONAL TRUE
PRINT INTL$("Hello cruel world!")
x = 2
PRINT x FORMAT NNTL$("There is %ld green bottle", "There are %ld green bottles",
x)
```

This program is saved as 'hello.bac'.

Step 2: compile program

Now compile the program using the '-x' option.

```
# bacon -x hello.bac
```

Next to the resulting binary, a *template* catalog file is created called 'hello.pot'.

Step 3: create catalog file

At the command line prompt, run the 'msginit' utility on the generated template file.
```
# msginit -l nl_NL -o hello.po -i hello.pot
```
In this example the nl_NL locale is used, which is Dutch. This will create a genuine catalog file called 'hello.po' from the template 'hello.pot'.

Step 4: add translations

Edit the catalog file 'hello.po' by adding the necessary translations.

Step 5: create object file

Again at the command line prompt, run the 'msgfmt' utility to convert the catalog file to a binary machine object file. The result will have the same name but with an '.mo' extension:
```
# msgfmt -c -v -o hello.mo hello.po
```

Step 6: install

Copy the resulting binary formatted catalog file 'hello.mo' into the correct locale directory. In this example, the locale used was 'nl_NL'. Therefore, it needs to be copied to the default textdomain directory '/usr/share/locale' appended with the locale name, thus: /usr/share/locale/nl_NL. In there, the subdirectory LC_MESSAGES should contain the binary catalog file.
```
# cp hello.mo /usr/share/locale/nl_NL/LC_MESSAGES/
```
The TEXTDOMAIN statement can be used to change the default directory for the catalog files.

Step 7: setup Unix environment

Finally, the Unix environment needs to understand that the correct locale must be used. To do so, simply set the LANG environment variable to the desired locale.
```
# export LANG nl_NL
```
After this, the BaCon program will show the translated strings.

Networking

TCP

With BaCon it is possible to create programs which have access to TCP networking. The following small demonstration shows a client program which fetches a website:
```
OPEN "www.basic-converter.org:80" FOR NETWORK AS mynet
SEND "GET / HTTP/1.1\r\nHost: www.basic-converter.org\r\n\r\n") TO mynet
REPEAT
    RECEIVE dat$ FROM mynet
    total$ = CONCAT$(total$, dat$)
UNTIL ISFALSE(WAIT(mynet, 5000))
CLOSE NETWORK mynet
PRINT total
```
The next program shows how to setup a TCP server which accepts multiple connections. It first imports the UNIX function 'fork'. The main program uses OPEN FOR SERVER multiple times. At each new connection the program forks itself and handles the incoming data:

```
IMPORT "fork" FROM "libc.so" TYPE int ALIAS "FORK"
PRINT "Connect from other terminals with 'telnet localhost 51000' and enter text
- 'quit' ends."
WHILE TRUE
    OPEN "localhost:51000" FOR SERVER AS mynet
    spawn = FORK()
    IF spawn = 0 THEN
        REPEAT
            RECEIVE dat$ FROM mynet
            PRINT "Found: ", dat$;
        UNTIL LEFT$(dat$, 4) = "quit"
        CLOSE SERVER mynet
        END
    ENDIF
WEND
```

UDP

The UDP mode can be set with the OPTION NETWORK statement. From then on a network program for UDP looks the same as a network program for TCP. This is an example client program:

```
OPTION NETWORK UDP
OPEN "localhost:1234" FOR NETWORK AS mynet
SEND "Hello" TO mynet
CLOSE NETWORK mynet
```

Example server program:

```
OPTION NETWORK UDP
OPEN "localhost:1234" FOR SERVER AS mynet
RECEIVE dat$ FROM mynet
CLOSE NETWORK mynet
PRINT dat$
```

BROADCAST

BaCon also knows how to send data in UDP broadcast mode. For example:

```
OPTION NETWORK BROADCAST
OPEN "192.168.1.255:12345" FOR NETWORK AS mynet
SEND "Using UDP broadcast" TO mynet
CLOSE NETWORK mynet
```

Example server program using UDP broadcast, listening to all interfaces:

```
OPTION NETWORK BROADCAST
OPEN "*:12345" FOR SERVER AS mynet
RECEIVE dat$ FROM mynet
CLOSE NETWORK mynet
PRINT dat$
```

MULTICAST

If UDP multicast is required then simply specify MULTICAST. Optionally, the TTL can be determined also. Here are the same examples, but using a multicast address with a TTL of 5:

```
OPTION NETWORK MULTICAST 5
OPEN "225.2.2.3:1234" FOR NETWORK AS mynet
SEND "This is UDP multicast" TO mynet
CLOSE NETWORK mynet
```

Example server program using multicast:

```
OPTION NETWORK MULTICAST
OPEN "225.2.2.3:1234" FOR SERVER AS mynet
RECEIVE dat$ FROM mynet
CLOSE NETWORK mynet
PRINT dat$
```

SCTP

BaCon also supports networking using the SCTP protocol. Optionally, a value for the amount of streams within one association can be specified.

```
OPTION NETWORK SCTP 5
OPEN "127.0.0.1:12380", "172.17.130.190:12380" FOR NETWORK AS mynet
SEND "Hello world" TO mynet
CLOSE NETWORK mynet
```

An example server program:

```
OPTION NETWORK SCTP 5
OPEN "127.0.0.1:12380", "172.17.130.190:12380" FOR SERVER AS mynet
RECEIVE txt$ FROM mynet
CLOSE NETWORK mynet
PRINT txt$
```

Ramdisks and memory streams

When creating programs which need heavy I/O towards the hard drive, it may come handy to create a ramdisk for performance reasons. Basically, a ramdisk is a storage in memory. While on Unix level administrator rights are required to create such a disk, BaCon can create an elementary ramdisk during runtime which is accessible within the program.

First, some amount of memory needs to be claimed which has to be opened in streaming mode. This returns a memory pointer which indicates the current position in memory, similar to a file pointer for files.

Then, the statements GETLINE and PUTLINE can be used to read and write lines of data towards the memory storage. For example:

```
memory_chunk = MEMORY(1000)
OPEN memory_chunk FOR MEMORY AS ramdisk
PUTLINE "Hello world" TO ramdisk
```

If the ramdisk needs to be read from the beginning, use MEMREWIND to reposition the memory pointer. In the next example, a GETLINE retrieves the line which was stored there:

```
MEMREWIND ramdisk
GETLINE text$ FROM ramdisk
```

If the option MEMSTREAM was set to TRUE, BaCon can treat the created ramdisk also as a string variable, which allows manipulations by using the standard string functions. The variable used for the memory pointer must be a string variable:

```
OPTION MEMSTREAM TRUE
memory_chunk = MEMORY(1000)
OPEN memory_chunk FOR MEMORY AS ramdisk$
PUTLINE "Hello world" TO ramdisk$
MEMREWIND ramdisk$
IF INSTR(ramdisk$, "world") THEN PRINT "found!
PRINT REPLACE$(ramdisk$, "Hello", "Goodbye")
```

Always make sure that there is enough memory to perform string changes to the ramdisk. The RESIZE statement safely can be used to enlarge the claimed memory during runtime, as this will preserve the data.

The contents of the ramdisk can be written to disk using PUTBYTE. However, it must be clear how many bytes need to be written, as the total amount of memory reserved to the ramdisk may be bigger than the actual amount of data. The function MEMTELL can be used in case the memory

pointer is positioned at the end of the ramdisk:

```
memory_chunk = MEMORY(1000)
OPEN memory_chunk FOR MEMORY AS ramdisk
    PUTLINE "Hello world" TO ramdisk
    OPEN "ramdisk.txt" FOR WRITING AS txtfile
        PUTBYTE memory_chunk TO txtfile CHUNK MEMTELL(ramdisk)-memory_chunk
    CLOSE FILE txtfile
CLOSE MEMORY ramdisk
FREE memory_chunk
```

Alternatively, if the ramdisk was opened with OPTION MEMSTREAM set to TRUE, the string function LEN also will return the length of the data.

Error trapping, error catching and debugging

BaCon can distinguish between 4 types of errors.

1. System errors. These relate to the environment in which BaCon runs.
2. Syntax errors. These are detected during the conversion process.
3. Compiler errors. These are generated by the C compiler and passed on to BaCon.
4. Runtime errors. These can occur during execution of the program.

When an error occurs, the default behaviour of a BaCon program is to stop. Only in case of runtime errors, it is possible to intercept the error with CATCH. This allows to proceed with a self-defined error handling function. This is especially convenient when creating GUI applications, as runtime errors by default appear on the Unix command prompt. To prevent BaCon detecting runtime errors altogether, use TRAP SYSTEM.

The reserved ERROR variable contains the number of the last error occurred. A full list of error numbers can be found in appendix A. With the ERR$ function a human readable text for the error number can be retrieved programmatically.

Next to these options, the statement TRACE ON can set the program in such a way that it is executed at each keystroke, step-by-step. This way it is possible to spot the location where the problem occurs. The ESC-key will then exit the program. To switch of trace mode within a program, use TRACE OFF.

Also the STOP statement can be useful in debugging. This will interrupt the execution of the program and return to the Unix command prompt, allowing intermediate checks. By using the Unix 'fg' command, or by sending the CONT signal to the PID of the program, execution can be resumed.

Notes on transcompiling

The process of translating a programming language into another language, and then compiling it, is also known as *transcompiling*. BaCon is a Basic to C translator, or a transcompiler, or transpiler.

When using BaCon, three stages can be distinguished:

1. conversion time
2. compilation time
3. runtime

It is important to realize that BaCon commands can function in all these stages. Examples of statements which have impact the on conversion stage are INCLUDE, RELATE, USEC, USEH, WITH and some of the OPTION arguments. These statements instruct BaCon about the way the Basic code should be converted.

A statement impacting the compilation stage is PRAGMA. With this statement it is possible to influence the behavior of the compiler.

Most other BaCon statements are effective during runtime. These form the actual program being executed.

It should be clear that the aforementioned stages cannot be mixed. For example, it is not possible to

define the argument for INCLUDE in a string variable, as the INCLUDE statement is effective during *conversion* time, while variables are used during *runtime*.
Note that except for system errors, the logic of the error messages basically follows the same structure: there are syntax errors (conversion time), compiler errors and runtime errors. The system errors do not fit in as they relate to the possibility of using BaCon itself.

Overview of BaCon statements and functions

ABS

ABS(x)
Type: function
Returns the absolute value of x. Example with and without ABS:

```
PRINT PI
PRINT ABS(PI)
```

ACOS

ACOS(x)
Type: function
Returns the calculated arc cosine 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> milliseconds. Use '0' to cancel an alarm. The alarm will interrupt any action the BaCon currently is performing; an alarm always has priority. Example:

```
SUB dinner
    PRINT "Dinner time!"
END SUB
ALARM dinner, 5000
```

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. Example:
```
IF x = 0 AND y = 1 THEN PRINT "Hello"
```

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")
```

ASIN

ASIN(x)
Type: function
Returns the calculated arcsine of x.

ATN

ATN(x)
Type: function
Returns the calculated arctangent of x.
```
PRINT ATN(1)
```

BREAK

BREAK [x]
Type: statement
Breaks out loop constructs like FOR/NEXT, WHILE/WEND or REPEAT/UNTIL.
The optional parameter can define to which level the break should take place in case of nested loops. This parameter should be an integer value higher than 0. See also CONTINUE to resume a loop.

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$

**CHOP$**(x$[, y$[, z]])
Type: function
Returns a string defined in x$ where on both sides <CR>, <NL>, <TAB> and <SPACE> have been removed. If other characters need to be chopped then these can be specified in the optional y$. The optional parameter z defines where the chopping must take place: 0 means on both sides, 1 means chop at the left and 2 means chop at the right. Examples:

```
PRINT CHOP$("bacon", "bn")
PRINT CHOP$(" hello world ", " ", 2)
```

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.

```
LET a$ = CHR$(0x23)
PRINT a$
```

CLEAR

CLEAR
Type: statement
Clears the terminal screen. To be used with ANSI compliant terminals.

CLOSE

CLOSE FILE|DIRECTORY|NETWORK|SERVER|MEMORY <handle>
Type: statement
Close file, directory, network or memory 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
```

Instead of color names, it is also possible to use their internal enumeration: black is 0, red is 1, green is 2, and so on. For BG a 0 can be used, and for FG a 1. For example:

```
COLOR 1 TO 3
PRINT "This is yellow!"
COLOR RESET
```

COLUMNS

COLUMNS
Type: function
Returns the amount of columns in the current ANSI compliant terminal. See also ROWS. Example:

```
PRINT "X,Y: ", COLUMNS, "," , ROWS
```

CONCAT$

**CONCAT$**(x$, y$, …)
Type: function
Returns the concatenation of x$, y$, ... The CONCAT$ function can accept an unlimited amount of arguments. Example:

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

The CONCAT$ function is in place for compatibility reasons. Instead, BaCon also accepts the '&' symbol as infix string concatenator. The following is the same example using '&':

```
txt$ = "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
```

CONTINUE

CONTINUE [x]
Type: statement
Skips the remaining body of loop constructs like FOR/NEXT, WHILE/WEND or REPEAT/UNTIL.
The optional parameter can define at which level a continue should be performed in case of nested loops, and should be an integer value higher than 0.

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.

COUNT

COUNT(string, y)
Type: function
Returns the amount of times the ASCII value <y> occurs in <string>. Example:

```
PRINT COUNT("Hello world", ASC("l"))
```

See also FILL$.

CURDIR$

CURDIR$
Type: function
Returns the full path of the current working directory.

CURSOR

CURSOR <**ON**|**OFF**> | <**FORWARD**|**BACK**|**UP**|**DOWN**> [x]
Type: statement
Shows ("on") or hides ("off") the cursor in the current ANSI compliant terminal. Also, the cursor can be moved one position in one of the four directions. Optionally, the amount of steps to move can be specified. Example:

```
PRINT "I am here"
CURSOR DOWN 2
PRINT "...now I am here"
```

DATA

DATA <x, y, z, ...>
Type: statement
Defines data. The DATA statement always contains data which is globally visible. 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. To start reading from the beginning again use RESTORE. Example:

```
DATA 1, 2, 3, 4, 5, 6
DATA 0.5, 0.7, 11, 0.15
DATA 1, "one", 2, "two", 3, "three", 4, "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> | [**ARRAY** <size>]
Type: statement
This statement is similar to the GLOBAL statement and is available for compatibility reasons.

DECR

DECR <x>[, y]
Type: statement

Decreases variable <x> with 1. Optionally, the variable <x> can be decreased with <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. Examples:

```
DEF FN func(x) = 3 * x
PRINT func(12)
DEF FN First$(x$) = LEFT$(x$, INSTR(x$, " ")-1)
PRINT First$("One Two Three")
```

DELETE

DELETE <**FILE**|**DIRECTORY**|**RECURSIVE**> <x$>
Type: statement
Deletes a file with the FILE argument, or an empty directory when using the DIRECTORY argument. The RECURSIVE argument can delete a directory containing files. It can also delete a complete directory tree. If an error occurs then this can be captured by using the CATCH statement.
Example:

```
DELETE FILE "/tmp/data.txt"
DELETE RECURSIVE "/usr/data/stuff"
```

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.

ENUM

ENUM
  item1, item2, item3
ENDENUM | **END ENUM**
Type: statement

Enumerates variables automatically. If no value is provided, the enumeration starts at 0 and will increase with integer numbers. Example:

```
ENUM
    cat, dog, fish
END ENUM
```

It is also possible to explicitly define a value:

```
ENUM
    Monday=1, Tuesday=2, Wednesday=3
END ENUM
```

EPRINT

EPRINT [value] | [text] | [variable] | [expression] [**FORMAT** <format>[**TO** <variable>[**SIZE** <size>]] | [,] | [;]
Type: statement
Same as PRINT but uses 'stderror' as output.

EQ

x **EQ** y
Type: operator
Verifies if x is equal to y. To improve readability it is also possible to use IS instead. Both the EQ and IS operators only can be used in case of numerical comparisons. Examples:

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

BaCon also accepts a single '=' symbol for comparison. Next to the single '=' also the double '==' can be used. These work both for numerical comparisons and for string comparisons. See also NE.

```
IF b$ = "Hello" THEN
    PRINT "world"
END IF
```

EQUAL

EQUAL(x$, y$)
Type: function
Compares two strings, and returns 1 if x$ and y$ are equal, or 0 if x$ and y$ are not equal. Use OPTION COMPARE to establish case insensitive comparison. Example:

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

The EQUAL function is in place for compatibility reasons. The following code also works:

```
IF 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$**(command$ [, stdin$])
Type: function
Executes an operating system command and returns the result to the BaCon program. The exit status of the executed command itself is stored in the reserved variable RETVAL. Optionally a second argument may be used to feed to STDIN. See SYSTEM to plainly execute a system command. Example:

```
result$ = EXEC$("ls -l")
result$ = EXEC$("bc", CONCAT$("123*456", NL$, "quit"))
```

EXIT

EXIT
Type: statement
Exits a SUB or FUNCTION prematurely. Note that functions which are supposed to return a value will return a 0. String functions will return an empty string.
Also note that it is allowed to write EXIT SUB or EXIT FUNCTION to improve code readability.

EXP

EXP(x)
Type: function
Returns e (base of natural logarithms) raised to the power of x.

EXTRACT$

**EXTRACT$**(x$, y$[, flag])
Type: function
Returns the string defined in <x$> from which the string mentioned in <y$> has been removed. The optional flag determines if the <y$> should be taken as a regular expression.
Examples:

```
PRINT EXTRACT$("bacon program", "ar")
PRINT EXTRACT$(name$, "e")
PRINT EXTRACT$("a b c", " .* ", TRUE)
```

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 ERR$ statement can be used to find out the error if TRAP is set to LOCAL. Example:

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

FILETIME

FILETIME(filename, type)
Type: function
Returns the timestamp of a file identified by <filename>, depending on the type of timestamp indicated in <type>. The type can be one of the following: 0 = access time, 1 = modification time and 2 = status change time. Example:

```
stamp = FILETIME("/etc/hosts", 0)
PRINT "Last access: ", MONTH$(stamp), " ", DAY(stamp), ", ",
YEAR(stamp)
```

FILETYPE

FILETYPE(filename)
Type: function
Returns the type of a file identified by <filename>. If an error occurs this function returns '0'. The

ERR$ statement can be used find out which error if TRAP is set to LOCAL. The following values may be returned:

Value	Meaning
0	Error or undetermined
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, ASC("@"))
```

See also COUNT to count the amount of times a character occurs in a string.

FLOOR

FLOOR(x)
Type: function
Returns the rounded down value of x. Note that this function always returns an integer value.

FOR

FOR var = x **TO** y [**STEP** z]
  <body>
  [**BREAK**]
NEXT [var]
FOR var$ **IN** source$ [**STEP** z$]
  <body>
  [**BREAK**]
NEXT [var]
Type: statement
With FOR/NEXT a body of statements can be repeated a fixed amount of times.
In the first usage the variable x will be increased until y with 1, unless a STEP is specified.

Example:
```
FOR x = 1 TO 10 STEP 0.5
    PRINT x
NEXT
```
In the second usage the variable x$ will get the space separated strings mentioned in source$.
Instead of a space, a set of characters can be specified as delimiter in the STEP keyword. Example:
```
FOR x$ IN "Hello cruel world"
    PRINT x$
NEXT
FOR y$ IN "1,2,3,4,5" STEP ","
    PRINT y$
NEXT
```

FP

FP (x)
Type: function
Returns the memory address of a function with name 'x'. Example:
```
SUB Hello
    PRINT "Hello world"
END SUB
DECLARE (*func)() TYPE void
func = FP(Hello)
CALL (*func)()
```

FREE

FREE x
Type: statement
Releases claimed memory (see also MEMORY). Example:
```
mem = MEMORY(500)
FREE mem
```
This statement also can be used to delete members from associative arrays:
```
FREE array$("abc")
```

FUNCTION

FUNCTION <name> ()|(STRING s, NUMBER i, FLOATING f, VAR v SIZE t)
  <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. Instead of the Bacon types STRING, NUMBER and FLOATING for the incoming arguments, also regular C-types can be used. With VAR a variable amount of arguments can be defined.
A FUNCTION always returns a value or a string, this should explicitly be specified with the RETURN statement. If the FUNCTION returns a string, then the function name should end with a '$' to indicate a string by value. Example:

```
FUNCTION fh2cel(NUMBER fahrenheit)
    LOCAL celsius
    celsius = fahrenheit*9/5 + 32
    RETURN celsius
END FUNCTION
FUNCTION Hello$(STRING name$)
    RETURN "Hello " & name$ & " !"
END FUNCTION
```

GETBYTE

GETBYTE <memory> **FROM** <handle> [**CHUNK** x] [**SIZE** y]
Type: statement
Retrieves binary data into a memory area from a either a file or a device identified by handle, with optional amount of <x> bytes depending on OPTION MEMTYPE (default amount of bytes = 1). Also optionally, the actual amount retrieved can be stored in variable <y>. Use PUTBYTE to write binary data.
Example program:

```
OPEN prog$ FOR READING AS myfile
    bin = MEMORY(100)
    GETBYTE bin 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. See also INPUT and WAIT.
Example:

```
PRINT "Press <escape> to exit now..."
key = GETKEY
IF key = 27 THEN
    END
```

```
END IF
```

GETLINE

GETLINE <variable$> **FROM** <handle>
Type: statement
Reads a line of data from a memory area identified by <handle> into a string variable. The memory area can be opened in streaming mode using the the OPEN statement (see also the chapter on ramdisks and memory streams). A line of text is read until the next newline character. Example:

```
GETLINE text$ FROM mymemory
```

See also PUTLINE to store lines of text into memory areas.

GETPEER$

GETPEER$(x)
Type: function
Gets the IP address and port of the remote host connected to a handle returned by OPEN FOR SERVER. Example:

```
OPEN "localhost:51000" FOR SERVER AS mynet
PRINT "Peer is: ", GETPEER$(mynet)
CLOSE SERVER mynet
```

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> | [**ARRAY** <size>]
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, meaning that variables declared with the GLOBAL keyword are visible in each part of the program. Use LOCAL for local declarations.
The ARRAY keyword is used to define a dynamic array, which can be resized with REDIM at a later stage in the program.
Optionally, within a SUB or FUNCTION it is possible to use GLOBAL in combination with RECORD to define a record variable which is visible globally.

```
GLOBAL x TYPE float
GLOBAL q$
GLOBAL new_array TYPE float ARRAY 100
GLOBAL name$ ARRAY 25
```

Multiple variables of the same type can be declared at once, using a comma separated list. In case of pointer variables the asterisk should be attached to the variable name:

```
GLOBAL x, y, z TYPE int
```

```
GLOBAL *s, *t TYPE long
```

GOSUB

GOSUB <label>
Type: statement
Jumps to a label defined elsewhere in the program (see also the LABEL statement). When a RETURN is encountered, the program will return to the last invoked GOSUB and continue from there. Note that a SUB or FUNCTION also limits the scope of the GOSUB; it cannot jump outside.
Example:

```
PRINT "Where are you?"
GOSUB there
PRINT "Finished."
END
LABEL there
    PRINT "In a submarine!"
    RETURN
```

GOTO

GOTO <label>
Type: statement
Jumps to a label defined elsewhere in the program. Note that a SUB or FUNCTION limits the scope of the GOTO; it cannot jump outside. See also the LABEL statement.

GOTOXY

GOTOXY x, y
Type: statement
Puts cursor to position x,y where 1,1 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.

HOST$

**HOST$**(name$)

Type: function
When name$ contains a hostname this function returns the corresponding IP address. If name$ contains an IP address the corresponding hostname is returned. If the name or IP address cannot be resolved an error is generated. Examples:

```
PRINT HOST$("www.google.com")
PRINT HOST$("127.0.0.1")
```

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** | **FI**
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 ENDIF or END IF or FI. 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
```

Use with care as nested IF/THEN statements using one function may confuse the parser.

IMPORT

IMPORT <function[(type arg1, type arg2, ...)]> **FROM** <library> **TYPE** <type> [**ALIAS** word]
Type: statement
Imports 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 "fork" FROM "libc.so" TYPE int ALIAS "FORK"
IMPORT "atan(double)" FROM "libm.so" TYPE double ALIAS
"arctangens"
```

INCLUDE

INCLUDE <filename>[, func1, func2, ...]
Type: statement
Adds a external BaCon file to current program. Includes may be nested. Optionally, it is possible to specify which particular functions in the included file need to be added. Examples:

```
INCLUDE "beep.bac"
INCLUDE "hug.bac", INIT, WINDOW, DISPLAY
```

INCR

INCR <x>[, y]
Type: statement
Increases variable <x> with 1. Optionally, the variable <x> can be increased with <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'.

```
position = INSTR("Hello world", "wo")
PRINT INSTR("Don't take my wallet", "all", 10)
```

INSTRREV

INSTRREV(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. The result is counted from the

beginning of haystack$. If not found then this function returns the value '0'.
See also OPTION STARTPOINT to return the result counted from the end of haystack$.

INTL$

**INTL$**(x$)
Type: function
Specifies that <x$> should be taken into account for internationalization. All strings which are surrounded by INTL$ will be candidate for the template catalog file. This file is created when BaCon is executed with the '-x' switch. See also the chapter about internationalization and the TEXTDOMAIN statement.

ISFALSE

ISFALSE(x)
Type: function
Verifies if x is equal to 0.

ISKEY

ISKEY(array, string)
Type: function
Returns TRUE (1) if <string> is defined as a key in the associative <array>. If not, FALSE (0) is returned. Example:

```
DECLARE array ASSOC int
array("hello") = 25
array("world") = 30
PRINT ISKEY(array, "goodbye")
PRINT ISKEY(array, "world")
```

ISTRUE

ISTRUE(x)
Type: function
Verifies if x is not equal to 0.

JOIN

JOIN <array> **BY** <sub> **TO** <string> **SIZE** <variable>
Type: statement
This statement can join elements of a one dimensional string array to a single string. The <sub> argument defines with which substring the elements are connected. The result is stored in <string>. The total amount of elements to be joined must be defined in <variable>. See also SPLIT to do the opposite. Example:

```
DECLARE name$[3]
name$[0] = "King"
name$[1] = "of"
```

```
name$[2] = "Holland"
JOIN name$ BY " " TO result$ SIZE 3
```

LABEL

LABEL <label>
Type: statement
Defines a label which can be jumped to by using a GOTO, GOSUB or CATCH GOTO statement. Also RESTORE may refer to a label. 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>] [**ARRAY** <size>]
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 variablename ends with a '$' then the variable will be a string.

The ARRAY keyword is used to define a dynamic array, which can be resized with REDIM at a later stage in the program.
Example:

```
LOCAL tt TYPE int
LOCAL q$
LOCAL new_array TYPE float ARRAY 100
LOCAL name$ ARRAY 25
```

Multiple variables of the same type can be declared at once, using a comma separated list. In case of pointer variables the asterisk should be attached to the variable name:

```
LOCAL x, y, z TYPE int
LOCAL *s, *t TYPE long
```

LOG

LOG(x)
Type: function
Returns the natural logarithm of x.

LOOKUP

LOOKUP <assoc> **TO** <array> **SIZE** <variable> [**STATIC**]
Type: statement
Retrieves all indexnames 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
```

The optional STATIC keyword allows the created <array> to be returned from a function.

MAKEDIR

MAKEDIR <directory>
Type: statement
Creates and empty directory. Parent directories are created implicitly. If the directory already exists then it is recreated. Errors like write permissions, disk quota issues and so on can be captured 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.

MEMCHECK

MEMCHECK(memory address)
Type: function
Verifies if <memory address> is accessible, in which case a '1' is returned. If not, this function returns a '0'. Example:

```
IF MEMCHECK(mem) THEN POKE mem, 1234
```

MEMORY

MEMORY(x)
Type: function
Claims memory of x size, returning a handle to the address where the memory block resides. Use FREE to release the memory. Note that OPTION MEMTYPE can influence the type of memory created. Example creating a memory area to store integers:

```
OPTION MEMTYPE int
area = MEMORY(100)
```

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. Example:

```
txt$ = "Hello cruel world"
PRINT MID$(txt$, 7, 5)
```

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. Instead, ISNOT can be used as well to improve code readability. The NE and ISNOT operators only work for numerical comparisons.
Next to these, BaCon also accepts the '!=' and '<>' constructs for comparison. These work both for numerical and string comparisons. See also EQ.

NL$

NL$
Type: variable
Represents the newline as a string.

NNTL$

**NNTL$**(x$, y$, value)
Type: function
Specifies that <x$> should be taken into account for internationalization. This is a variation to INTL$. With NNTL$ singularities and multitudes can be specified, which are candidate for the template catalog file. This file is created when BaCon is executed with the '-x' switch. See also TEXTDOMAIN and INTL$ and the chapter on internationalization. Example:

```
LET x = 2
PRINT x FORMAT NNTL$("There is %ld green bottle\n", "There are %ld
```

```
green bottles\n", x)
```

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** [**FROM** address[:port]]|**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. Optionally, the source IP address and port can be specified using FROM.

```
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 network connections. When invoked multiple times in TCP mode using the same host and port, OPEN SERVER will not create a new socket, but accept another incoming connection. Instead of specifying an IP address, also the Unix wildcard '*' can be used to listen to all interfaces. See also OPTION NETWORK to set the network protocol.

```
OPEN "*: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$
```

When used with DEVICE, a file or device can be opened in any mode. The open mode can set by using OPTION DEVICE. Use PUTBYTE or GETBYTE to write and retrieve data from the opened device.

```
OPEN "/dev/ttyUSB0" FOR DEVICE AS myserial
SETSERIAL myserial SPEED B38400
GETBYTE mem FROM myserial CHUNK 5 SIZE received
CLOSE DEVICE myserial
```

OPTION

OPTION <**BASE** x> | <**COMPARE** x> | <**SOCKET** x> | <**NETWORK** type [ttl]> | <**MEMSTREAM** x> | <**MEMTYPE** type> | <**COLLAPSE** x> | <**INTERNATIONAL** x> | <**STARTPOINT** x> | <**DEVICE** 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 and LOOKUP statements. It has no impact on arrays which assign their values statically at the moment of declaration.
- The COMPARE argument defines if string comparisons or regular expressions with REGEX should be case sensitive (0) or not (1). The default is *case sensitive* (0).
- The SOCKET argument defines the timeout for setting up a socket to an IP address. Default value is 5 seconds.

- The NETWORK argument defines the type of protocol: TCP, UDP, BROADCAST, MULTICAST or SCTP. When MULTICAST is selected also an optional value for TTL can be specified. When SCTP is selected an optional value for the amount of streams can be specified. Default setting for this option is: TCP. Default value for TTL is 1. Default amount of SCTP streams is 1.
- The MEMSTREAM argument allows the handle created by the OPEN FOR MEMORY statement to be used as a string variable (1). Default value is 0.
- The MEMTYPE argument defines the type of memory to be used by POKE, PEEK, MEMORY, RESIZE, PUTBYTE and GETBYTE. Default value is 'char' (1 byte). Any valid C type can be used here, for example 'float', 'unsigned int', 'void' etc.
- The COLLAPSE argument specifies if the results of the SPLIT statement may contain empty results (0) in case the separator occurs as a sequence in the target string, or not (1). Default value is 0.
- The INTERNATIONAL argument enables support for internationalization of strings. It sets the textdomain for INTL$ and NNTL$ to the current filename. See also TEXTDOMAIN and the chapter on creating internationalization files. The default value is 0.
- The STARTPOINT argument has impact on the way the INSTRREV function returns its results. When set to 1, the result of the INSTRREV function is counted from the end of the string. Default value is 0 (counting from the beginning of the string).
- The DEVICE argument determines the way a device or file is opened in the OPEN FOR DEVICE statement. By default BaCon uses the following open mode: O_RDWR|O_NOCTTY|O_SYNC. Other common Unix open modes are O_APPEND, O_ASYNC, O_CREAT, O_EXCL, O_NONBLOCK and O_TRUNC. Please refer to the open manpage for more details on the open modes.

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 value stored at memory address x. The type of the returned value can be determined with OPTION MEMTYPE.

PI

PI

Type: variable
Reserved variable containing the number for PI: 3.1415926536.

POKE

POKE <x>, <y>
Type: statement
Stores a value <y> at memory addres <x>. Use PEEK to retrieve a value from a memory address. Use OPTION MEMTYPE to determine the type of the value to store. Example:

```
OPTION MEMTYPE float
mem = MEMORY(500)
POKE mem, 32.123
```

POW

POW(x, y)
Type: function
Raise x to the power of y.

PRAGMA

PRAGMA <**OPTIONS** x> | <**LDFLAGS** x> | <**COMPILER** x> | <**INCLUDE** x>
Type: statement
Instead of passing commandline arguments to influence the behavior of the compiler, it is also possible to define these arguments programmatically. Mostly these arguments are used when embedding variables or library dependent structures into BaCon code. Example when SDL code is included in the BaCon program:

```
PRAGMA LDFLAGS SDL
PRAGMA INCLUDE SDL/SDL.h
```

Example when GTK2 code is included in the BaCon program:

```
PRAGMA LDFLAGS `pkg-config --cflags --libs gtk+-2.0`
PRAGMA INCLUDE gtk-2.0/gtk/gtk.h
PRAGMA COMPILER gcc
```

Example on passing optimization parameters to the compiler:

```
PRAGMA OPTIONS -O2 -s
```

Multiple arguments can be passed too:

```
PRAGMA LDFLAGS iup cd iupcd im
PRAGMA INCLUDE iup.h cd.h cdiup.h im.h im_image.h
```

PRINT

PRINT [value] | [text] | [variable] | [expression] [**FORMAT** <format>][**TO** <variable> [**SIZE** <size>]] | [,] | [;]
Type: statement
Prints a numeric value, text, variable or result from expression to standard output. See EPRINT for printing to stderr. 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 ", 42, " years which is ", 12 + 30 FORMAT "%s%d%s
%d\n"
```
The result also can be printed to a string variable. This can also be done in combination with FORMAT. To achieve this, use the keyword TO. Optionally, the total amount of resulting characters can be provided with the SIZE keyword. If no size is given, BaCon will use its default internal buffer size (512 characters).
```
PRINT "Hello cruel world" TO hello$
PRINT "Hello" & "cruel" & "world" TO hello$ SIZE 32
t = NOW + 300
PRINT HOUR(t), MINUTE(t), SECOND(t) FORMAT "%.2ld%.2ld%.2ld" TO
time$
PRINT MONTH$(t) FORMAT "%s" TO current$ SIZE 15
```

PROTO

PROTO <function name>[,function name [, ...]] [**ALIAS** word]
Type: statement
Defines an external function so it is accepted by the BaCon parser. Mutliple 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. See PUSH to push values to the stack.

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.
```
' Initially create a new 0 value for stack
```

```
' This will only be 0 when stack wasn't declared before
PULL stack
PUSH stack
' Increase and push the stack 2x
' Stack has now 3 values
INCR stack
PUSH stack
PUSH "End"
PULL var$
' Print and pull current stack value - will return "end" 1 0
PRINT var$
PULL stack
PRINT stack
PULL stack
PRINT stack
```

PUTBYTE

PUTBYTE <memory> **TO** <handle> [**CHUNK** x] [**SIZE** y]
Type: statement
Store binary data from a memory area to either a file or a device identified by handle, with an optional amount of <x> bytes, depending on OPTION MEMTYPE (default amount of bytes = 1). Also optionally, the actual amount stored can be captured in variable <y>.
This statement is the inverse of GETBYTE, refer to this command for an example.

PUTLINE

PUTLINE "text"|<variable$> **TO** <handle>
Type: statement
Write a line of string 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 (see also the chapter on ramdisks and memory streams). Example:

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

See also GETLINE to retrieve a line of text from a memory area.

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 <x1[, x2, x3, ...]>
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
```

Also, multiple variables may be provided:

```
READ a, b, c, d$
DATA 10, 20, 30, "BaCon"
```

See RESTORE to define where to start reading the data.

READLN

READLN <var> **FROM** <handle>
Type: statement
Reads a line of ASCII data from a file identified by <handle> into variable <var>. See the GETBYTE statement to read binary data. Example:

```
READLN txt$ FROM myfile
```

RECEIVE

RECEIVE <var> **FROM** <handle> [**CHUNK** <chunksize>] [**SIZE** <amount>]
Type: statement
Reads data from a network location identified by handle into a string variable or memory area. Subsequent reads return more data until the network buffer is empty. The chunk size can be determined with the optional CHUNK keyword.
The amount of bytes actually received can be retrieved by using the optional SIZE keyword. If the amount of bytes received is 0, then the other side has closed the connection in an orderly fashion. In such a situation the network connection needs to be reopened. 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
```

REDIM

REDIM <var> **TO** <size>
Type: statement
Redimensions a one dimensional dynamic array to a new size. The contents of the array will be preserved. If the array becomes smaller then the elements at the end of the array will be cleared. The dynamic array has to be declared previously using DECLARE or LOCAL. Example:

```
REDIM a$ TO 20
```

REGEX

REGEX (txt$, expr$)
Type: function
Applies a POSIX Extended Regular Expression expr$ to the string txt$. If the expression matches, the position of the first match is returned. If not, this function returns '0'. The length of the last match is returned in the reserved variable REGLEN.
Use OPTION COMPARE to set case sensitive matching. Examples:

```
' Does the string match alfanum 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")
```

REGLEN

REGLEN
Type: variable
Reserved variable containing the length of the last REGEX match.

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.
BaCon also accepts C-style block comments: this can be done by surrounding multiple lines using /* and */.

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$, replacement$ [, flag])
Type: function
Substitutes a substring <needle$> in <haystack$> with <replacement$> and returns the result. The replacement does not necessarily need to be of the same size as the substring. The optional flag determines if the <needle$> should be taken as a regular expression.
Examples:
```
PRINT REPLACE$("Hello world", "l", "p")
PRINT REPLACE$("Some text", "me", "123")
PRINT REPLACE$("Goodbye <all>", "<.*>", "123", TRUE)
PRINT REPLACE$("abc123def", "[[:digit:]]", "x", TRUE)
```

RESIZE

RESIZE <x>, <y>
Type: statement
Resizes memory area starting at address <x> to an amount of <y> of the type determined by OPTION MEMTYPE. If the area is enlarged, the original contents of the area remain intact.

RESTORE

RESTORE [label]
Type: statement
Restores the internal DATA pointer(s) to the beginning of the first DATA statement.
Optionally, the restore statement allows a label from where the internal DATA pointer needs to be restored. See also READ. Example:

```
DATA 1, 2, 3, 4, 5
LABEL txt
DATA "Hello", "world", "this", "is", "BaCon"
RESTORE txt
READ dat$
```

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
```

RETURN

RETURN [value]
Type: statement
If RETURN has no argument it will return to the last invoked GOSUB. If no GOSUB was invoked previously then RETURN has no effect.
Only in case of functions the RETURN statement must contain a value. This is the value which is returned when the FUNCTION is finished.

RETVAL

RETVAL
Type: variable
Reserved variable containing the return status of the operating system commands executed by SYSTEM or EXEC$.

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. See also the function RANDOM for a more convenient way of generating random numbers. Example:

```
SEED NOW
x = RND
```

ROUND

ROUND(x)
Type: function
Rounds x to the nearest integer number. For compatibility reasons, the keyword INT may be used instead. Note that this function always returns an integer value.
See also FLOOR to round down to the nearest the integer and MOD to get the fraction from a fractional number.

ROWS

ROWS
Type: function
Returns the amount of rows in the current ANSI compliant terminal. Use COLUMNS to get the amount of columns.

SCROLL

SCROLL <**UP** [x]|**DOWN** [x]>
Type: statement
Scrolls the current ANSI compliant terminal up or down one line. Optionally, the amount of lines to scroll can be provided.

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 and RANDOM 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 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 "Alien detected"
END SELECT
```

SEND

SEND <var> **TO** <handle> [**CHUNK** <chunk>] [**SIZE** <size>]
Type: statement
Sends data in <var> to a network location identified by <handle>. Optionally, the amount of bytes to send can be specified with the CHUNK keyword. As by default SEND will consider the <var> to be a string, the default amount of data is the string length of <var>. However, instead of a string, also binary data can be sent by using a memory area created by the MEMORY function. In such a situation it is obligatory to also specify the chunk size.
The amount of bytes actually sent can be retrieved by using the optional SIZE keyword. For an example of SEND, 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"
```

SETSERIAL

SETSERIAL <device> **IMODE**|**OMODE**|**CMODE**|**LMODE**|**SPEED**|**OTHER** <value>
Type: statement
This statement can set the properties of a serial device. The Input Mode (IMODE), Output Mode (OMODE), Control Mode (CMODE) and Local Mode (LMODE) can be set, as well as the speed and the special properties on the serial device. A discussion on the details of all these options is outside the scope of this manual. Please refer to the TermIOS documentation of your C compiler instead.
Example usage opening a serial port in 8N1, ignoring 0-byte as a break, canonical, and non-blocking with a timeout of 0.5 seconds:

```
OPEN "/dev/ttyUSB0" FOR DEVICE AS myserial
SETSERIAL myserial SPEED B9600
SETSERIAL myserial IMODE ~IGNBRK
SETSERIAL myserial CMODE ~CSIZE
SETSERIAL myserial CMODE CS8
SETSERIAL myserial CMODE ~PARENB
SETSERIAL myserial CMODE ~CSTOPB
SETSERIAL myserial LMODE ICANON
SETSERIAL myserial OTHER VMIN = 0
SETSERIAL myserial OTHER VTIME = 5
```

SGN

SGN(x)
Type: function
Returns the sign of x. If x is a negative value, this function returns -1. If x is a positive value, this function returns 1. If x is 0 then a 0 is returned.

SIN

SIN(x)
Type: function
Returns the calculated SINUS of x.

SIZEOF

SIZEOF(type)
Type: function
Returns the bytesize of a C type.

SLEEP

SLEEP <x>
Type: statement

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

SORT

SORT <x> [**SIZE** <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. The amount of elements involved can be specified with SIZE. This keyword is optional for static arrays, but should always be used in case of dynamic arrays. Also 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> [**STATIC**]
Type: statement
This statement can split a string into smaller pieces. The <sub> argument determines where the string is being split. The results are stored in <array>. As sometimes 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
```

The above example will return four elements, of which the third element is empty. If OPTION COLLAPSE is put to 1, the above example will return three elements, ignoring empty entries. See also JOIN.
The optional STATIC keyword allows the created <array> to be returned from a function.

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, VAR v SIZE t)]
  <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. With VAR a variable amount of arguments can be defined. Example:

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

SWAP

SWAP x, y
Type: statement
Swaps the contents of the variables x and y.

SYSTEM

SYSTEM <command$>

Type: statement
Executes an operating system command. It causes the BaCon program to hold until the command has been completed. The exit status of the executed command itself is stored in the reserved variable RETVAL. 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 tangent of x.

TELL

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

TEXTDOMAIN

TEXTDOMAIN <domain$>, <directory$>
Type: statement
When OPTION INTERNATIONAL is enabled, BaCon by default configures a textdomain with the current filename and a base directory "/usr/share/locale" for the message catalogs. With this statement it is possible to explicitly specify a different textdomain and base directory.

TIMER

TIMER
Type: function
Keeps track of the amount of milliseconds the current program is running. Example:

```
iter = 1
WHILE iter > 0 DO
    IF TIMER = 1 THEN BREAK
    INCR iter
WEND
PRINT "Got ", iter-1, " iterations in 1 millisecond!"
```

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 displayed on 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, BaCon performs error trapping (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.
When set to SYSTEM, error 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
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$

**UCASE$**(x$)
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 put unmodified into the generated C source file. Example:

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

USEH

USEH
  <body>
ENDUSEH | END USEH
Type: statement
Defines a body with C declarations and/or definitions. This code is put unmodified into the generated global header source file. This can particularly be useful in case of using variables from external libraries. See also USEC to pass C source code. Example:

```
USEH
    char *str;
    extern int pbl_errno;
END USEH
```

VAL

VAL(x$)
Type: function
Returns the actual value of x$. This is the opposite of STR$. Example:

```
nr$ = "456"
q = VAL(nr$)
```

VAR

VAR <array$> **SIZE** <x>
Type: statement
Declares a variable argument list in a FUNCTION or SUB. There may not be other variable declarations in the function header. The arguments to the function are put into an array of strings, and the resulting amount of elements is stored in <x>. Example:

```
OPTION BASE 1
SUB demo (VAR arg$ SIZE amount)
    LOCAL x
    PRINT "Amount of incoming arguments: ", amount
    FOR x = 1 TO amount
        PRINT arg$[x]
    NEXT
END SUB

' No argument
demo(0)
```

```
' One argument
demo("abc")
' Three arguments
demo("123", "456", "789")
```

VERSION$

VERSION$
Type: variable
Reserved variable which contains the BaCon version text.

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.
This statement also can be used to find out if a key is pressed without actually waiting for a key, so without interrupting the current program. In this case, use the STDIN filedescriptor (0) as the handle. Example:

```
REPEAT
    PRINT "Press Escape... waiting..."
    key = WAIT(STDIN_FILENO, 50)
UNTIL key = 27
```

As can be observed in this code, instead of '0' the reserved POSIX variable STDIN_FILENO can be used also. See also appendix B for more standard POSIX variables.

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)
```

WEEKDAY$

WEEKDAY$(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 = 41
    .street = Falkwood Area 1
    .city = The Hague
END WITH
```

WRITELN

WRITELN "text"|<var> **TO** <handle>
Type: statement
Write a line of ASCII data to a file identified by handle. Refer to the PUTBYTE statement to write binary data. 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

Code	Meaning
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
31	Gosub nesting too deep
32	Could not open device
33	Error configuring serial port
34	Error accessing device
35	Error in INPUT

Appendix B: standard POSIX variables

Variable	Value
EXIT_SUCCESS	0
EXIT_FAILURE	1
STDIN_FILENO	0
STDOUT_FILENO	1
STDERR_FILENO	2
RAND_MAX	System dependent

Please report errors to: `REVERSE$("gro.retrevnoc-cisab@retep")`
Created with LibreOffice 3.6.2.
Back to top of document