BaCon 4.8 documentation

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 code can be compiled using generic C compilers like GCC or CC. It can be compiled using a C++ compiler as well.
BaCon intends 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 package and run the installer. The converter can be used as follows:

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

Note that BASH 4.x or higher is required to execute the Shell script version of BaCon.
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.sh myprog.bac
```

Alternatively, also Kornshell93 (releases after 2012) or Zshell (versions higher than 4.x) can be used:

```
ksh ./bacon.sh myprog.bac
zsh ./bacon.sh 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.sh 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.sh 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 (defaults to 'gcc')
- -l: pass a library to the C linker
- -o: pass a compiler option 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 (defaults to the current directory)
- -x: extract gettext strings from generated c sources
- -z: allow the usage of lowercase statements and functions

- -e: starts the embedded ASCII editor
- -f: create a shared object of the program
- -n: do not compile the C code automatically after conversion
- -y: suppress warning about temporary files if these exist
- -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)
- -q: suppress line counting during conversion and only show summary after conversion
- -r: convert and execute the resulting program in one step
- -s: suppress warnings about semantic errors
- -t: tweak internal BaCon parameters for the string core engine and memory pool:
  pbc=<x>: set the Pool Block Count to <x> defaults to: 1024
  pbs=<x>: set each individual Pool Block Size to <x> defaults to: 1024
  hld=<x>: set the Hash Linear Depth probing to <x> defaults to: 16
  hss=<x>: set the Hash String Store size to <x> defaults to: 0x100000
  mrb=<x>: set the Maximum Return Buffers to <x> defaults to: 64
- -w: store command line settings in the configuration file ~/.bacon/bacon.cfg. 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

The shell script implementation can convert and compile the BaCon version of BaCon. This will deliver the binary version of BaCon which has an extremely high conversion performance. On newer systems, the average conversion rate usually lies above 10.000 lines per second.
This documentation refers both to the shell script and binary implementation of BaCon.
Here are a few examples showing the usage of command line parameters:

- Convert and compile program with debug symbols: bacon -o -g program.bac
- Convert and compile program, optimize and strip: bacon -o -O2 -o -s program.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 program.bac
- Convert and compile program including an additional C header file: bacon -i header.h yourprogram.bac
- Store compile options permanently: bacon -w -l tcmalloc -o -O2 program.bac (subsequent invocations of BaCon will now always use the mentioned options)

Most of the aforementioned options also can be used programmatically by use of the PRAGMA keyword.

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 a space and the '\' symbol at the end of the line. The LET statement may be omitted, so a line may contain an assignment only. Expressions are not converted, but are passed unchanged to the C compiler (*lazy conversion*).
BaCon does not require line numbers. More statements per line are accepted. These should be separated by the colon symbol ':'.
All keywords must be written in capitals to avoid name clashes with existing C keywords or functions from libc. Keywords in small letters are considered to be variables unless the '-z' command line option is specified, in which case BaCon tries to parse lowercase keywords as if they were written in capitals. Note that this may lead to unexpected results, for example if the program uses variable names which happen to be BaCon keywords.
Statements are always written without using brackets. Functions however 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.
Subroutines may be defined using SUB/ENDSUB and do not return a value. With the FUNCTION/ENDFUNCTION statements a function can be defined which does return a value. The return value must be explicitly stated with the statement RETURN.
The three main variable types in BaCon are defined as STRING, NUMBER and FLOATING. These are translated to char*, long and double.
A variable will be declared implicitly when the variable is used in an assignment (e.g. LET) or in a statement which assigns a value to a variable. By default, implicitly declared variables are of 'long' type. This default can be changed by using the OPTION VARTYPE statement. Note that 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 implicitly declared within a SUB or FUNCTION also by default have a global scope. When declared with the LOCAL statement variables will have a scope local to the FUNCTION or SUB.
In case of implicit assignments, BaCon assumes numeric variables to be of long type, unless specified otherwise with OPTION VARTYPE. Also, it is possible to define a variable to any other C-type explicitly using the DECLARE and LOCAL statements.
Next to this, BaCon accepts type suffixes as well. For example, if a variable name ends with the '$' symbol, a string variable is assumed. If a variable name ends with the '#' symbol, a float variable is assumed. If a variable name ends with the '%' symbol, it is considered to be an integer variable. The type suffixes also can be used when defining a function name.

Mathematics, variables

The standard C operators for mathematics can be used, like '+' for addition, '-' for subtraction, '/' for division and '*' for multiplication. For the binary 'and', the '&' symbol must be used, and for the binary 'or' use the pipe symbol '|'. Binary shifts are possible with '>>' and '<<'.

C operator	Meaning	C Operator	Meaning
+	Addition	|	Inclusive or

-	Subtraction	^	Exclusive or
*	Multiplication	>>	Binary shift right
/	Division	<<	Binary shift left
&	Binary and		

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
0	Equal to	String, numeric
!=, <>	Not equal to	String, numeric
>	Greater than	String, numeric also allows GT
<	Less than	String, numeric also allows LT
>=	Greater or equal	String, numeric also allows GE
<=	Less or equal	String, numeric also allows LE
EQ, IS	Equal to	Numeric
NE, ISNOT	Not equal to	Numeric
AND, OR	Logical and,or	String, numeric
BETWEEN	In between	String, numeric
BEYOND	Outside	String, numeric
EQUAL()	Equal to	String

The BETWEEN and BEYOND keywords

When using equations in WHILE, IF, REPEAT, IIF or IIF$ it often happens that a check needs to be performed to see if a certain value lies within a range. For this purpose, BaCon accepts the BETWEEN comparison keyword. For example:

```
IF 5 BETWEEN 0;10 THEN PRINT "Found"
```

This comparison will return TRUE in case the value 5 lies within 0 and 10. The comparison will include the lower and upper boundary value during evaluation. Note that the lower and upper values are being separated by a semicolon. Alternatively, the keyword AND may be used here as well:

```
IF 5 BETWEEN 0 AND 10 THEN PRINT "Found"
```

Note that this may lead to confusing constructs when adding more logical requirements to the same equation.

The BETWEEN comparison also accepts strings:

```
IF "C" BETWEEN "Basic" AND "Pascal" THEN PRINT "This is C"
```

The order of the mentioned range does not matter, the following code will deliver the exact same

result:
```
IF "C" BETWEEN "Pascal" AND "Basic" THEN PRINT "This is C"
```
In case the boundary values should be excluded, BaCon accepts the optional EXCL keyword:
```
IF variable BETWEEN 7 AND 3 EXCL THEN PRINT "Found"
```
The last example will print "Found" in case the variable lies within 3 and 7, values which themselves are excluded.
Similarly, to check if a value lies outside a certain range, the keyword BEYOND can be used:
```
IF 5 BEYOND 1 AND 3 THEN PRINT "Outside"
```
Note that in case of the BEYOND keyword the boundary values themselves are considered to be part of the outside range. The same EXCL keyword can be used to exclude them.

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 their size can be redimensioned during runtime. The latter can be achieved with the REDIM statement. This is only possible for arrays with one dimension.
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 such an array also can be expressed by a variable. Furthermore, the size of a one dimensional dynamic array can be changed afterwards with the REDIM statement. This statement also works for implicitly created dynamic arrays in the SPLIT and LOOKUP statements.

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.
The UBOUND function returns the dimension of an array. In case of multi-dimensional arrays the total amount of elements will be returned. The UBOUND function works for static and dynamic arrays, but also for associative arrays.

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 only works for 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. Therefore only dynamic arrays can be returned.
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, LOOKUP, COLLECT, PARSE and MAP 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.
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
```

As with other variables, declaring associative arrays within a function using LOCAL will ensure a local scope of the array.
An associative array can have any amount of dimension. The indexes in an associative array should be separated by a comma. For example:

```
DECLARE demo$ ASSOC STRING
demo$("one") = "hello"
demo$("one", "two") = "world"
```

Alternatively, the indexes also can be specified in a delimited string format, using a single space as delimiter:

```
demo$("one two") = "world"
```

Note that the OPTION BASE statement has no impact on associative arrays. Also note that an associative array cannot be part of a RECORD structure.
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. It also is possible to copy the contents of one associative array to another. This can simply be done by using the assignment operator, as follows:

```
array$() = other$()
```

Next to this, the actual index names in an associative array can be looked up using the LOOKUP statement. This statement returns a dynamically created array containing all string indexes. The size of the resulting array is dynamically declared as it depends on the amount of available elements. Instead of creating a dynamic array, it is also possible to return the indexes of an associative array into a delimited string by using the function OBTAIN$.

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).

The function NRKEYS will return the amount of members in an associative array.

Deleting individual associative array members can be done by using the FREE statement. This will leave the associative array insertion order intact. The FREE statement also can be used to delete a full associative array in one step.

The function INDEX$ allows looking up a specific key based on value. The function INVERT can swap the keys and values of an associative array. The SORT statement also can sort associative arrays based on their value, effectively changing the insertion order in the underlying hash table.

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
```

Records

Declaration

Records are collections of variables which belong together. A RECORD has a name by itself and members of the record can be accessed by using the <name>.<member> notation. The members should be declared using the LOCAL statement. For example:

```
RECORD rec
    LOCAL value
    LOCAL nr[5]
END RECORD
rec.value = 99
```

As soon a record is created, it also exists as a type. The name of the type always consists of the record name followed by the '_type' suffix. From then on, it is possible to declare other variables as being of the same type. To continue with the same example:

```
DECLARE var TYPE rec_type
var.value = 123
```

Arrays of records

Record definitions also can be created as static arrays or as dynamic arrays. The size of the static array is determined during compile time and the data will be stored in the stack frame of a SUB or FUNCTION. This means that the array data is lost when the SUB or FUNCTION is ended. Example of a static array definition:

```
RECORD data[10]
    LOCAL info$
END RECORD
```

To declare a dynamic array of records, the keyword ARRAY must be used. The size of a dynamic record array is determined during runtime, and therefore, can be set with variables and functions. The data is stored in the heap. The BaCon memory management will clean up the data when leaving a FUNCTION or SUB. Example:

```
RECORD data ARRAY 10
    LOCAL name$[5]
    LOCAL age[5]
END RECORD
```

Note that dynamic arrays of records do not allow members which are dynamic arrays themselves.

Passing records to functions or subs

To pass a record, simply declare the variable name with the appropriate record type in the header of the function or sub. Example code:

```
RECORD rec
    LOCAL nr
    LOCAL area$
END RECORD
```

```
SUB subroutine(rec_type var)
    PRINT var.nr
    PRINT var.area$
ENDSUB
rec.nr = 123
rec.area$ = "europe"
CALL subroutine(rec)
```

Similarly, it is possible to pass an array of records as well. Note the square brackets in the function header:

```
RECORD rec ARRAY 10
    LOCAL nr
    LOCAL area$
END RECORD
SUB subroutine(rec_type var[])
    PRINT var[0].nr
    PRINT var[0].area$
ENDSUB
rec[0].nr = 123
rec[0].area$ = "europe"
CALL subroutine(rec)
```

Returning records from functions

In order to return a record from a function, the record type must be visible to the caller. The below example declares the record in the main program. The function declares a variable of the same type and initializes the record to 0. This initialization is obligatory for string members to work properly. Then some values are assigned. Lastly, the complete record is returned to the caller:

```
RECORD rec
    LOCAL id
    LOCAL zip$[2]
END RECORD
FUNCTION func TYPE rec_type
    LOCAL var = { 0 } TYPE rec_type
    var.id = 1
    var.zip$[0] = "XJ342"
    var.zip$[1] = "YP198"
    RETURN var
ENDFUNCTION
rec = func()
PRINT rec.id
PRINT rec.zip$[0]
PRINT rec.zip$[1]
```

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.)
Note that as soon an existing string variable is referred to by a reference variable, the string will not profit from the optimized high performance string engine anymore.

ASCII, Unicode, UTF8

BaCon is a byte oriented converter. This means it always will assume that a string consists of a sequence of ASCII bytes. Though this works fine for plain ASCII strings, it will cause unexpected results in case of non-Latin languages, like Chinese or Cyrillic. However, BaCon supports UTF8 encoded strings also.
The original text already may contain the UTF8 byte order mark 0xEF 0xBB 0xBF. The function HASBOM can be used to detect if such byte order mark is present. To add or delete a byte order mark, use EDITBOM$.
In order to work with UTF8 strings, OPTION UTF8 needs to be enabled. This option will put all string related functions in UTF8 mode at the cost of some performance loss in string processing. Next to this option, BaCon also provides a few functions which relate to UTF8 encoding. The following functions work independently from OPTION UTF8:

- ULEN will correctly calculate the actual characters based on the binary UTF8 sequence.
- BYTELEN will show the actual amount of bytes used by a UTF8 string.
- ISASCII can be used to verify if a string only consists of ASCII data.
- UTF8$ needs the Unicode value as argument and returns the corresponding character depending on environment settings and the current font type.
- UCS needs a UTF8 character as an argument and returns the corresponding Unicode value.
- ESCAPE$ will convert a UTF8 string to an ASCII sequence with escape characters.
- UNESCAPE$ will convert an ASCII sequence with escaped characters back to valid UTF8.
- HASBOM will detect if the UTF8 byte order mark is present in the text
- EDITBOM$ can be used to add or delete a UTF8 byte order mark

---

Binary trees

BaCon has a built-in API for binary trees. Compared to arrays, a binary tree is a data structure which can access its elements in a faster and more efficient manner. Regular arrays store an element

in a linear way, which, in worst case, can end up in long sequential lookup times. A binary tree however uses an internal decision tree to lookup an element, of which the lookup time, depending on the tree position, is logarithmic.
A typical application using a binary tree is a database, which needs to lookup information from a large amount of data. A search in a binary tree will be a lot faster compared to a plain linear search in a regular array.
To declare a binary tree, BaCon uses the DECLARE or LOCAL keyword together with TREE. For example, to declare a binary tree containing strings:

```
DECLARE mytree TREE STRING
```

Subsequently, it is possible to declare other types as well, for example integers or floats:

```
DECLARE myinttree TREE int
DECLARE myfloattree TREE float
```

After the declaration, new values (nodes) can be added to the tree. Adding a string is very straightforward. It can be added to a binary tree using the TREE statement:

```
TREE mytree ADD "hello"
text$ = "world"
TREE mytree ADD text$
```

Similarly, to add an integer or float to the binary tree:

```
TREE myinttree ADD 567
TREE myfloattree ADD 4.127
```

When adding a duplicate string or value nothing happens. Such attempt will silently be ignored. As a result, all entries in a binary tree are unique.
The FIND function can lookup the presence of an element. If found, the FIND function will return TRUE (1), otherwise it will return FALSE (0). The following example looks up a string in a binary tree:

```
IF FIND(mytree, "abc") THEN PRINT "Found!"
```

Similarly it is possible to lookup an integer or float value:

```
result = FIND(myfloattree, 2.2)
var = 123
result = FIND(myinttree, var)
```

It is also possible to remove an element from the tree using the DELETE statement. If an element is not found then nothing happens:

```
DELETE result$ FROM mytree
DELETE 1.2 FROM myfloattree
```

The COLLECT statement can collect all the strings or values of all nodes in the binary tree and put them into a regular array:

```
COLLECT mytree TO allnodes$
COLLECT myinttree TO intnodes
```

Lastly, the function TOTAL can be used to find out the total amount of elements in a binary tree:

```
PRINT "Amount of nodes:", TOTAL(mytree)
```

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 explain how 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 several 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.

Remarks

In case global dynamic string arrays are used by the BaCon shared object, then these need to be initialized prior to using the arrays. This can be done by calling a special function available in each shared object created in BaCon: the 'BaCon_init()' function. In case the shared object is compiled by a GNU C compatible compiler, then this function is executed automatically.

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' manually, 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

Using 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 over HTTP:

```
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$ = total$ & dat$
UNTIL ISFALSE(WAIT(mynet, 5000))
CLOSE NETWORK mynet
PRINT total$
```

The following program verifies if a remote site can be reached by a specific port, trying to access it via a specific interface on the localhost:

```
CATCH GOTO Error
OPEN "www.basic-converter.org:443" FOR NETWORK FROM "192.168.1.107" AS net
PRINT "The host 'basic-converter.org' listens at port 443."
CLOSE NETWORK net
END
LABEL Error
    PRINT "The host 'basic-converter.org' either is not reachable, filtered or
has port 443 not open."
```

The next program shows how to setup a simple TCP server. The main program uses OPEN FOR SERVER after which the ACCEPT function handles the incoming connection:

```
PRINT "Connect from other terminals with 'telnet localhost 51000' and enter text
- 'quit' ends."
OPEN "localhost:51000" FOR SERVER AS mynet
WHILE TRUE
  fd = ACCEPT(mynet)
    REPEAT
        RECEIVE dat$ FROM fd
        PRINT "Found: ", dat$;
    UNTIL LEFT$(dat$, 4) = "quit"
    CLOSE SERVER fd
WEND
```

UDP

The UDP mode can be set with the OPTION NETWORK statement. After this, 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 SERVER 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 SERVER 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 SERVER 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 SERVER mynet
PRINT txt$
```

TLS secured network connections

The previous chapter demonstrated network connections where the data is transferred over the wire in plain text. However, with the increasing vulnerabilities in current network traffic, it usually is a good idea to apply Transport Layer Security (TLS).
BaCon does not implement a propriety TLS standard by its own. However, it can make use of existing libraries like OpenSSL. Alternatively, BaCon also allows other TLS implementations, in case these provide an OpenSSL compatible API. Examples are the GnuTLS and WolfSSL libraries but also projects forking from OpenSSL, like BoringSSL and LibreSSL.
To enable TLS, simply add OPTION TLS to the program. From then on, new network connections are considered to be TLS encapsulated:

```
OPTION TLS TRUE
```

This option enables the usage of OpenSSL by default. The presence of the OpenSSL libraries and header files on the system is required. BaCon will try to convert the source program and assumes the default locations of the OpenSSL development files. However, if these reside at a different location, it is possible to specify their location as follows:

```
PRAGMA TLS openssl INCLUDE <openssl/ssl.h> LDFLAGS -lssl -lcrypto
```

The OPTION TLS statement is always required, but instead of OpenSSL, it is possible to specify a different library:

```
PRAGMA TLS gnutls
```

This will provide an indication that BaCon should make use of the development files from the GnuTLS implementation. If these files should be taken from a special location:

```
PRAGMA TLS gnutls INCLUDE <gnutls/openssl.h> LDFLAGS -lgnutls -lgnutls-openssl
```

Lastly, BaCon supports WolfSSL as well:

```
PRAGMA TLS wolfssl
```

Also for the WolfSSL library it is possible to specify the location of the development files:

```
PRAGMA TLS wolfssl INCLUDE <wolfssl/options.h> <wolfssl/openssl/ssl.h> LDFLAGS -
lwolfssl
```

BaCon can use the function CA$ to discover the certificate authority of the connection, and CN$ to discover the common name. The CIPHER$ function can be used to obtain details on the encryption and the VERIFY function to verify the validity of the certificate.

The following small program queries a Mac address API over TLS using default OpenSSL:

```
OPTION TLS TRUE
website$ = "api.macvendors.com"
mac$ = "b0:52:16:d0:3c:fb"
OPEN website$ & ":443" FOR NETWORK AS mynet
SEND "GET /" & mac$ & " HTTP/1.1\r\nHost: " & website$ & "\r\n\r\n" TO mynet
RECEIVE info$ FROM mynet
CLOSE NETWORK mynet
PRINT TOKEN$(info$, 2, "\r\n\r\n")
```

The next program shows how to setup a simple webserver using TLS:

```
OPTION TLS TRUE
CERTIFICATE "key.pem", "certificate.pem"
CATCH GOTO resume_on_error
CONST Msg$ = "<html><head>Hello from BaCon!</head></html>"
PRINT "Connect with your browser to 'https://localhost:51000'."
OPEN "localhost:51000" FOR SERVER AS mynet
WHILE TRUE
    client = ACCEPT(mynet)
    IF client < 0 THEN CONTINUE
    RECEIVE dat$ FROM client
    PRINT dat$
    SEND "HTTP/1.1 200 Ok\r\nContent-Length: " & STR$(LEN(Msg$)) & "\r\n\r\n" &
Msg$ TO client
    CLOSE SERVER client
WEND
LABEL resume_on_error
    RESUME
```

The program below demonstrates a plain HTTPS connection using GnuTLS:

```
OPTION TLS TRUE
PRAGMA TLS gnutls INCLUDE <gnutls/openssl.h> LDFLAGS -lgnutls -lgnutls-openssl
website$ = "www.google.com"
OPEN website$ & ":443" FOR NETWORK AS mynet
SEND "GET / HTTP/1.1\r\nHost: " & website$ & "\r\n\r\n" TO mynet
WHILE WAIT(mynet, 2000)
    RECEIVE data$ FROM mynet
    total$ = total$ & data$
    IF REGEX(data$, "</html>") THEN BREAK
WEND
PRINT REPLACE$(total$, "\r\n[0-9a-fA-F]+\r\n", "\r\n", TRUE)
PRINT "--------------------------"
PRINT CIPHER$(mynet)
PRINT CA$(mynet)
PRINT CN$(mynet)
PRINT VERIFY(mynet, pem_file_with_rootca$)
PRINT "--------------------------"
CLOSE NETWORK mynet
```

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.

Delimited strings

A delimited string is a string which can be cut into parts, based on a character or on a set of characters. An example of such a string is a plain space delimited line in a textbook, where the words are separated by a whitespace. Another example is an ASCII file, in which the lines are separated by a newline. A very famous example of a delimited string is the Comma Separated Value (CSV) string. From another point of view, a delimited string also can be looked at as a list of items, which is the basis of LISP like languages.
The SPLIT statement can be used to split a string into elements of an array, based on a delimiter. As with all statements and functions handling delimited strings, the SPLIT statement will ignore a delimiter when it occurs between double quotes. Such delimiter is considered to be part of the string. For example:
```
csv$ = "This,is,a,CSV,string,\"with,an\",escaped,delimiter"
SPLIT csv$ BY "," TO member$ SIZE x
```
One of the resulting members of the array will contain `"with,an"` because the comma is enclosed

within double quotes. BaCon will consider this a piece of text where the characters should be kept together. The behavior of skipping delimiters within double quotes can be changed by setting or unsetting OPTION QUOTED. The JOIN statement can be used to merge array elements back into one (delimited) string.

Instead of SPLIT, it is possible to use FOR..IN as well. This statement will subsequently return the parts of the delimited text into a variable. Example:

```
FOR i$ IN "aa bb cc"
```

In this example, the variable 'i$' will subsequently have the value 'aa', 'bb' and 'cc' assigned. Also the FOR statement will skip a delimiter occurring within double quotes. Note that the OPTION COLLAPSE will prevent empty results, both for SPLIT and FOR.

It is also possible to return a single member in a delimited string. This can be achieved with the TOKEN$ function. Note that all element counting is 1-based. The following returns the 5th member of a space delimited string, being `"e f"`. It does not use an optional third parameter, because for all delimited string processing, BaCon defines the default delimiter as a single space:

```
PRINT TOKEN$("a b c d \"e f\" g h i j", 5)
```

But this function also works in case some other delimiter is used. The delimiter must then be specified in the third optional argument.

```
PRINT TOKEN$("1,2,3,4,5", 3, ",")
```

All the functions handling delimited strings accept such an optional argument. Alternatively, OPTION DELIM can define the delimiter string which should be used in subsequent functions. As mentioned, the default value is a single space.

The function EXPLODE$ will return a delimited string based on a specified amount of characters:

```
PRINT EXPLODE$("aabbcc", 1)
```

Alternatively, the inline loop function COIL$ can create a delimited string also, using an optional variable, for example the alphabet:

```
PRINT COIL$(i, 26, CHR$(64+i))
```

The MERGE$ function will do the opposite: merging the elements of a delimited string to one regular string, again optionally specifying a delimiter, for example:

```
PRINT MERGE$("aa,bb,cc", ",")
```

The ISTOKEN function can verify if a text occurs as a token in a delimited string. If so, this function returns the actual position of the token:

```
t$ = "Kiev Amsterdam Lima Moscow Warschau Vienna Paris Madrid Bonn Bern Rome"
PRINT "Is this a token: ", ISTOKEN(t$, "Rome")
```

To obtain the first members from a delimited string, the function HEAD$ can be used:

```
PRINT "The first 2 elements: ", HEAD$(t$, 2)
```

Similarly, it is possible to get the last elements by using TAIL$:

```
PRINT "The last element: ", TAIL$(t$, 1)
```

The TAIL$ and HEAD$ functions have their complementary functions in LAST$ and FIRST$. The following example will show all members of a delimited string *except* the first 2 members:

```
PRINT "All except the first 2 elements: ", LAST$(t$, 2)
```

The next code shows all members *except* the last:

```
PRINT "All except the last element: ", FIRST$(t$, 1)
```

It also is possible to obtain an excerpt using CUT$. The following piece of code will get the members from delimited string 't$' starting at position 2 and ending at position 4 inclusive:

```
PRINT "Some middle members: ", CUT$(t$, 2, 4)
```

Instead of fetching a member, BaCon also can change a member in a delimited string directly by using the CHANGE$ function:

```
result$ = CHANGE$("a,b,c,d,e,f,g,h,i,j", 5, "Ok", ",")
```

It is even possible to swap two members in a delimited string with the EXCHANGE$ function:
```
result$ = EXCHANGE$("a b c d e f g h i j", 5, 4)
```
The UNIQ$ function will return a delimited string where all members occur only once:
```
city$ = "Kiev Lima Moscow \"New York\" Warschau \"New York\" Rome"
PRINT "Unique member cities: ", UNIQ$(city$)
```
To add more members to a delimited string, use APPEND$:
```
t$ = APPEND$(t$, 2, "Santiago")
```
And to delete a member from a delimited string, use DEL$:
```
t$ = DEL$(t$, 3)
```
There are also functions to sort the members in a delimited string (SORT$) and to put them in reversed order (REV$). With PROPER$ it is possible to capitalize the first letter of each individual element in a delimited string. The ROTATE$ function rotates the items in a delimited string. The COLLAPSE$ function will remove empty items in a delimited string. The MATCH function can compare elements between two delimited strings and PARSE can return parts of a delimited string based on wildcards. The WHERE function returns the actual character position of the indicated token.
To determine if a string contains a delimiter at all, the HASDELIM function can be used, while the DELIM$ function can change the actual delimiter in a string to some other definition.
If a member still contains double quotes and escaped double quotes, then this can be flattened out by using the FLATTEN$ function. This function will remove double quotes and put escaping one level lower:
```
PRINT FLATTEN$("\"Hello \\\" world\"")
```
Lastly, the function AMOUNT will count the number of members in a delimited string:
```
nr = AMOUNT("a b c d e f g h i j")
PRINT AMOUNT("a,b,c,d,e,f,g,h,i,j", ",")
```
BaCon also has string functions available to handle delimited strings which use unbalanced delimiters. These are delimiters which consist of different characters, or different sets of characters. Examples of such strings are HTML or XML strings. They can be handled by functions like INBETWEEN$ and OUTBETWEEN$ very easily. For example, to obtain the title of a website from an HTML definition:
```
PRINT INBETWEEN$("<html><head><title>Website</title></head>", "<title>",
"</title>")
```
By default, INBETWEEN$ will perform a non-greedy match, but the fourth optional argument can be set to specify a greedy match.
Similarly, the OUTBETWEEN$ function will return everything but the matched substring, effectively cutting out a substring based on unbalanced delimiters.
Note that OPTION COLLAPSE does not impact both INBETWEEN$ and OUTBETWEEN$.

Regular expressions

BaCon can digest POSIX compliant regular expressions when using the REGEX function or the string functions EXTRACT$, REPLACE$ and WALK$. For this, BaCon relies on the standard libc implementation. However, it is possible to define a different regular expression engine with the PRAGMA statement.
For example, to specify the very fast NFA based regular expression library TRE, the following line must be added at the top of the program:
```
PRAGMA RE tre
```

This will include the header file from the TRE library and will link against its shared object. Of course, the system needs to have the required development files from the TRE library installed. BaCon will add the default locations of all necessary files to the compile flags. In case these files are kept at a different location, it is possible to define this explicitly as well:

```
PRAGMA RE tre INCLUDE <tre/regex.h> LDFLAGS -ltre
```

Next to the TRE library, also the Oniguruma library can be specified:

```
PRAGMA RE onig
```

When specifying the required development files:

```
PRAGMA RE onig INCLUDE <onigposix.h> LDFLAGS -lonig
```

Also the famous PCRE library is supported:

```
PRAGMA RE pcre
```

The full definition looks like:

```
PRAGMA RE pcre INCLUDE <pcreposix.h> LDFLAGS -lpcreposix
```

Basically, any regular expression library with a functional POSIX interface can be specified. This allows a lot of flexibility when certain features for regular expression parsing are required. The libraries TRE, Oniguruma and PCRE do not need a further INCLUDE or LDFLAGS specification if their development files have their default names and reside at their default location.

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 behavior of a BaCon program is to stop. Only in case of runtime errors, it is possible to let the program handle the error.

- In case of *statements,* the CATCH GOTO command can jump to 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.
- In case of *functions,* the OPTION ERROR must be set to FALSE to prevent the program from stopping. The program then needs to check the reserved ERROR variable to handle any unexpected situation.

Alternatively, it is possible to set a callback function for both statements and functions. This callback function can be defined by the CATCH ERROR statement. It should point to a function with three arguments: the first argument capturing the statement or function causing the error, the second the name of the file and the last the line number.

To prevent BaCon from 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 does not make sense to define the argument for the INCLUDE statement 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 relate to the possibility of using BaCon itself.

Using the BaCon spartanic editor (BaSE)

BaCon comes with a limited built-in ASCII editor which can display BaCon code using syntax highlighting. The editor requires an ANSI compliant terminal. To start the editor, simply use the '-e' argument and a filename:

```
# bacon -e prog.bac
```

If the file does not exist, then an empty screen occurs. Pressing the <ESC> button will pop up a menu down below the screen. The options are:
- (H)elp: display the Help screen
- (Q)uit: quit the editor

The usual functionality for editing text applies. Most actions can be performed using the <CTRL> key and a regular key. The <CTRL>+<h> key combination will pop up the Help screen.
The Help screen will show the following information:
- <CTRL>+<n>: new file
- <CTRL>+<l>: load file
- <CTRL>+<s>: save file
- <CTRL>+<x>: cut line of text
- <CTRL>+<c>: copy line of text
- <CTRL>+<v>: paste line of text
- <CTRL>+<w> or <CTRL>+<del>: delete line

- <HOME>: put cursor at start of line
- <END>: put cursor at end of line
- <PgUp>: move one page upwards
- <PgDn>: move page downwards
- <cursor keys>: navigate through text
- <CTRL>+<f>: find term in code
- <CTRL>+<g>: goto line number
- <CTRL>+<e>: compile and execute program
- <CTRL>+<a>: apply indentation
- <CTRL>+<r>: toggle line numbers
- <CTRL>+<b>: toggle text boldness
- <CTRL>+<d>: show context info
- <CTRL>+<h>: show this help
- <CTRL>+<q>: quit BaCon spartanic editor

The context info works by first moving the cursor to the statement or function for which more information is desired. Then, the <CTRL>+<d> key combination will try to lookup the keyword in the manpage.

It is also possible to adjust the colors of the syntax highlighting. Create the BaCon configuration file ~/.bacon/bacon.cfg if it does not exist yet. The following keywords can set the coloring scheme, following the numbering of the COLOR statement:

- statement_color (default: 2)
- function_color (default: 6)
- variable_color (default: 3)
- type_color (default: 3)
- number_color (default: 1)
- comment_color (default: 4)
- quote_color (default: 5)
- default_color (default: 7)

For example, to set the color for quoted text strings to green, the following definition can be added to the BaCon configuration file:

```
quote_color 2
```

Support for GUI programming

BaCon has a few functions available which enable basic GUI programming. These functions follow the "object/property" model and implement simple event handling.

Enabling GUI functions

To enable these functions, add the following line to your code:

```
OPTION GUI TRUE
```

By default, BaCon assumes Xaw as a backend. Though this is not the most attractive widget set in the world, it is always available on any Unix platform which has X installed. Alternatively, a different backend can be set using PRAGMA GUI. Currently, Xaw3d, Motif, TK, GTK2, GTK3, GTK4 and the console based CDK widgets (experimental) sets are supported. The following code will select Motif as a backend:

```
PRAGMA GUI motif
```

Instead of "motif", the PRAGMA GUI statement also accepts "xaw3d", "uil", "gtk2", "gtk3", "gtk4", "tk" and "cdk".

BaCon generates the source code for the GUI so the required header files for the toolkits should be available on the compiling platform.

Defining the GUI

To define the GUI, the function GUIDEFINE can be used. This function may occur in the program only once. Its string argument should contain a list of declarative widget definitions, each set of definitions grouped between curly brackets. The GUIDEFINE function returns the GUI id number. This is an example for the Xaw widget toolkit:

```
gui = GUIDEFINE( " \
{ type=window name=window callback=window XtNwidth=300 XtNtitle=\"Information\"
} \
{ type=dialogWidgetClass name=dialog parent=window XtNlabel=\"Enter term:\"
XtNvalue=\"<term>\" } \
{ type=commandWidgetClass name=submit parent=dialog callback=XtNcallback
XtNlabel=\"Submit\" } ")
```

As can be observed, each widget refers to properties which are specific to the Xaw toolkit. A widget must have a type and a name. Optionally, a widget can be attached to a parent, and it can submit a callback signal, in which case the signal name must be specified.

The callback keyword can also define a particular string to be returned in the event loop, defined after the signal name. The following will return "click" when the button is pressed:

```
{ type=commandWidgetClass name=submit parent=dialog callback=XtNcallback,click
XtNlabel=\"Submit\" }
```

Setting properties

The function GUISET can be used to specify more properties at a later stage in the program:

```
CALL GUISET(gui, "label", XtNjustify, XtJustifyLeft)
```

The function GUIGET is used to retrieve a value from a widget. The value is stored in a pointer variable:

```
CALL GUIGET(gui, "text", XtNstring, &txt)
```

For the TK backend, these functions can get and set variables from and to the several TK functions.

Entering the mainloop

Once the GUI is defined, the program can enter the event loop using GUIEVENT$. It requires the GUI id as an argument:

```
WHILE TRUE
    event$ = GUIEVENT$(gui)
    SELECT event$
        CASE "submit"
            BREAK
    ENDSELECT
```

```
WEND
```

When an event has happened, either the name of the widget causing the event is returned, or the defined string in the callback keyword. The BaCon program can decide what to do next.

In some cases it may be necessary to obtain a value which is passed to the callback by the widget library. In such situation, the GUIEVENT$ function accepts an optional second boolean parameter. This will add the incoming callback value as a pointer attached as a string to the return value.

For example, to obtain the selected item in a XawList widget, the Xaw library returns a struct containing information about the XawList. This can be fetched as follows:

```
WHILE TRUE
    event$ = GUIEVENT$(gui, TRUE)
    SELECT TOKEN$(event$, 1)
        CASE "submit"
            BREAK
        CASE "list"
            info = (XawListReturnStruct*)DEC(TOKEN$(event$, 2))
            PRINT TOKEN$(info->string, 1)
    ENDSELECT
WEND
```

Defining helper functions

It is also possible to define supplementary helper functions. A widget library can implement additional functions to perform actions on widgets. These helper functions can be configured in the BaCon program by setting up a function pointer and then use GUIFN.

For example, to define a helper function showing a Xaw widget:

```
LOCAL (*show)() = XtPopup TYPE void
CALL GUIFN(id, "window", show, XtGrabNonexclusive)
```

Such definition may seem unnecessary, however, using GUIFN has several advantages: it is compliant with the overall API design, the source code becomes smaller in size, and most importantly, we do not need to worry about argument types of the helper function.

For the TK backend, the GUIFN statement can define additional TCL code in the current TK context.

Using native functions

The function GUIWIDGET will return the memory address of a widget based on the defined name for that widget. This can come handy in cases where a GUI helper function is used natively.

Overview of BaCon statements and functions

ABS

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

where the latter always will produce a positive output:

```
PRINT x-y
PRINT ABS(x-y)
```

ACCEPT

ACCEPT(fd)
Type: function
In a network program, this function waits for an incoming connection and returns a new descriptor to be used for SEND and RECEIVE. If the ACCEPT functions fails, then the returned value is a negative number. Example:

```
OPEN "localhost:51000" FOR SERVER AS mynet
WHILE TRUE
    fd = ACCEPT(mynet)
    REPEAT
        RECEIVE dat$ FROM fd
        PRINT "Found: ", dat$;
    UNTIL LEFT$(dat$, 4) = "quit"
    CLOSE SERVER fd
WEND
```

ACOS

ACOS(x)
Type: function
Returns the calculated arc cosine of x, where x is a value in radians.

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. The value '0' will cancel an alarm. The alarm will interrupt any action the BaCon program currently is performing; an alarm always has priority. After the sub is executed, the program will continue the operation it was doing when the alarm occurred. 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 "printf" TO DISPLAY
DISPLAY("Hello world\n")
```

ALIGN$

**ALIGN$**(string$, width, type [,indent])
Type: function
Aligns a multi line <string$> over a maximum of <width> characters. The <type> indicates the kind of alignment to apply: 0 = left alignment, 1 = right alignment, 2 = center alignment, and 3 means fill or justify.
The alignment takes place in three stages. First, if the text starts with the UTF-8 byte order mark bytes 0xEF 0xBB 0xBF, then these are removed and the ALIGN$ function will automatically enable UTF8 mode for the text. Then, if the original text contains newline characters (0x0A), these are replaced with a single space. However, empty lines (double new lines indicating a paragraph) are preserved, as well as all other special characters, like a space (0x20), tab (0x09), carriage return (0x0D), non breaking space (0xA0) or a form feed (0x0C). Therefore, in some cases, it may be necessary to remove such special characters before using ALIGN$.
The second stage will try to find the best spot where to replace the space character (0x20) for a newline character (0x0A). This is done within the provided <width>. If there are redundant adjacent spaces then these are removed.
Note that the ALIGN$ function will not hyphenate words. Lines are being cut at a white space where possible. If a word does not fit in the provided width by itself, then it will be wrapped around.
The third stage will apply the chosen type of alignment. In case type is 0, 1 or 2, the lines in the final result are being padded with a single space character. In case type is 3, additional spaces are being added equally in between the words to align the text on both sides, except for the last line in a paragraph (where paragraphs are considered to be separated from each other by an empty line).
If the original text contained the UTF8 order mark 0xEF 0xBB 0xBF, then the ALIGN$ function will put back the byte order mark in the first bytes of the result. For more information on the UTF8 byte order mark, see also the HASBOM and EDITBOM$ functions.
The optional argument <indent> will prepend additional space characters to each line. Example:

```
data$ = LOAD$("ascii_data.txt")
PRINT ALIGN$(data$, 40, 0)
```

The ALIGN$ can handle UTF8 strings correctly as well. If the original text does not contain the UTF8 byte order mark then UTF8 mode should be enabled manually. The following example aligns a UTF8 text without byte order mark at two sides, each line not containing more than 50 characters, starting 10 positions from the left, while the text is being stripped from carriage return symbols:

```
OPTION UTF8 TRUE
text$ = LOAD$("Jane_Austen.txt")
```

```
PRINT ALIGN$(EXTRACT$(text$, CR$), 50, 3, 10)
```

AMOUNT

AMOUNT(string$ [,delimiter$])
Type: function
Returns the amount of tokens in a string split by delimiter$. The delimiter$ is optional. If it is omitted, then the definition from OPTION DELIM is assumed. When specified, it may consist of multiple characters. If delimiter$ occurs between double quotes in string$ then it is ignored.
Example:

```
PRINT AMOUNT("a b c d \"e f\" g h i j")
PRINT AMOUNT("Dog Cat@@@Mouse Bird@@@123@@@456@@@789", "@@@")
```

AND

<expr> **AND** <expr>
Type: operator
Performs a logical 'and' between two expressions. For the binary 'and', use the '&' symbol. Example:

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

APPEND

APPEND string$ **TO** filename$
APPEND string$, other$
Type: statement
This statement can be used two ways. The first is to save a string to disk in one step. If the file already exists, the data will be appended. See BAPPEND for appending binary files in one step, and OPEN/WRITELN/READLN/CLOSE to read and write to a file using a filehandle in append mode. In the second way, this statement simply will append one string to another, thereby modifying the original string.
Examples:

```
APPEND result$ TO "/tmp/more_data.txt"
APPEND str$, "world"
```

APPEND$

**APPEND$**(string$, pos, token$ [, delimiter$])
Type: function
Inserts <token$> into a delimited string$ split by delimiter$, at position <pos>. The delimiter$ is optional. If it is omitted, then the definition from OPTION DELIM is assumed. When specified, it may consist of multiple characters.
If the <pos> parameter is 0, or is bigger than the amount of members in <string$>, then <token$> is appended. If OPTION COLLAPSE is FALSE (default) then APPEND$ always will add a delimiter to an empty <string$>, after which <token$> will be added. In case of TRUE however, appending <token$> will simply copy the contents of <token$> to an empty <string$>.
If the <pos> parameter is negative, then <string$> will be returned unmodified. If delimiter$ occurs

between double quotes in string$, then it is ignored. This behavior can be changed by setting OPTION QUOTED to FALSE. See DEL$ to delete members, and the chapter on delimited string functions for more information about delimited strings. Example:

```
PRINT APPEND$("Rome Amsterdam Kiev Bern Paris London", 2,
"Vienna")
```

ARGUMENT$

ARGUMENT$
Type: variable
Reserved variable containing name of the program and the arguments to the program. These are all separated by spaces.
If the CMDLINE function is used then this variable will contain optional arguments to command line functions.

ASC

ASC(char)
Type: function
Calculates the ASCII value of char (opposite of CHR$). See also UCS for UTF8 characters.
Example:

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

ASIN

ASIN(x)
Type: function
Returns the calculated arcsine of x, where x is a value in radians.

ATN

ATN(x)
Type: function
Returns the calculated arctangent of x, where x is a value in radians.

```
PRINT ATN(RAD(90))
```

ATN2

ATN2(y, x)
Type: function
Returns the calculated arctangent of y/x. The sign of the arguments is used to determine the quadrant.

```
PRINT ATN2(30, -35)
```

B64DEC$

**B64DEC$**(x$)
Type: function
Returns the decoded string or data from a BASE64 string. In case of binary data, the LEN function will correctly provide the amount of bytes available in the string. Example:

```
PRINT B64DEC$("QmFDb24gaXMgZnVu")
```

Example where binary PNG data is recovered into a string variable and saved:

```
png$ = B64DEC$(some_encoded_png$)
BSAVE png$ TO "picture.png" SIZE LEN(png$)
```

B64ENC$

**B64ENC$**(x$[, length])
Type: function
Returns the encoded string from a regular string or memory area. In case of a string, the length of the string is the default. In case of a memory area, the additional <length> argument must specify the amount of bytes to encode. Examples:

```
PRINT B64ENC$("Encode me")
PRINT B64ENC$(mem, 1024)
```

BAPPEND

BAPPEND data **TO** filename$ **SIZE** amount
Type: statement
Saves a memory area with binary data to disk in one step. If the file already exists, the data will be appended. See APPEND for appending text files in one step, and OPEN/PUTBYTE/GETBYTE/CLOSE to read and write to a file using a filehandle. Example:

```
BAPPEND mem TO "/home/me/data" SIZE 10
```

BASENAME$

**BASENAME$**(filename$ [, flag])
Type: function
Returns the filename part of a given full filename. The optional [flag] indicates the part of the filename to be returned. The values are: 0 = full filename (default), 1 = filename without extension and 2 = extension without filename. See also DIRNAME$.

BIN$

BIN$(x)
Type: function
Calculates the binary value of x and returns a string with the result. The type size depends on the setting of OPTION MEMTYPE. If MEMTYPE is set to char (default), then 8 bits are returned, if it is set to short then 16 bits are returned, etc. See also DEC to convert back to decimal.

BIT

BIT(x)
Type: function
This function returns the value for a bit on position <x>. If x = 0 then it returns 1, if x = 1 then it returns 2, if x = 2 then it returns 4 and so on.

```
PRINT BIT(x)
```

BLOAD

BLOAD(filename$)
Type: function
Performs a load into a memory address of a binary file. The memory address is returned when the loading was successful. When done with the data, the memory should be freed with the FREE statement. See LOAD$ for loading text files in one step, and OPEN/PUTBYTE/GETBYTE/CLOSE to read and write to a file using a file handle. Example:

```
binary = BLOAD("/home/me/myprog")
PRINT "First two bytes are: ", PEEK(binary), " ", PEEK(binary+1)
FREE binary
```

BREAK

BREAK [x]
Type: statement
Breaks out loop constructs like FOR/NEXT, WHILE/WEND, REPEAT/UNTIL or DOTIMES/DONE.
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.

BSAVE

BSAVE data **TO** filename$ **SIZE** amount
Type: statement
Saves a memory area with binary data to disk in one step. If the file already exists it is overwritten. The amount must be specified in bytes. See SAVE for saving text files in one step, and OPEN/PUTBYTE/GETBYTE/CLOSE to read and write to a file using a filehandle. Example:

```
BSAVE mem TO "/home/me/picture.png" SIZE 12123
```

BYTELEN

BYTELEN(x$, y [, z])
Type: function
Returns the actual byte length of UTF8 string x$ in case of y characters. This is a wrapper function which can be used in combination with regular string functions, allowing correct processing of UTF8 string sequences. If the optional argument z is set then start counting the byte length from the right size of string x$. Example:

```
str$ = "A © and a ® symbol"
PRINT LEFT$(str$, BYTELEN(str$, 3))
PRINT RIGHT$(str$, BYTELEN(str$, 8, TRUE))
```

CA$

CA$(connection)
Type: function
Returns the Certificate Authority from the certificate used in the current network connection. Assumes that TLS has been enabled. See the chapter on secure network connections for more details. See also CN$.

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 fh2celsius(72) TO celsius
PRINT celsius
```

CATCH

CATCH GOTO <label> | **RESET** | **ERROR** <function>
Type: statement
The GOTO keyword sets the error function where the program should jump to if runtime error checking is enabled with TRAP. This only is applicable for statements. For an example, see the RESUME statement.
The RESET keyword restores the BaCon default error messages for statements.
The ERROR keyword allows setting a callback function where the program will jump to in case an error occurs. This works both for statements and functions provided that OPTION ERROR is set to FALSE. This is to prevent that erroneous functions will stop the program. The callback function should have three arguments which will hold the name of the statement or function, the name of the file and the line number where the error occurred. Example:

```
OPTION ERROR FALSE
CATCH ERROR help
SUB help(c$, f$, no)
    PRINT "Error is: ", ERR$(ERROR), " in function ", c$, " in
file '", f$, "' at line ", no
    PRINT "Callback ended"
END SUB
```

CEIL

CEIL(x)

Type: function
Rounds x up to the nearest integral (integer) number. This function always returns a float value. See also FLOOR and ROUND.

CERTIFICATE

CERTIFICATE <key.pem>, <certificate.pem>
Type: statement
Defines the private key and certificate file in PEM format. This function is used to setup TLS server networking. For an example, see the chapter on TLS.

CHANGE$

**CHANGE$**(string$, position, new$ [, delimiter$])
Type: function
Changes the token in string$, which is split by delimiter$, at position with new$. The delimiter$ is optional. If it is omitted, then the definition from OPTION DELIM is assumed. When specified, it may consist of multiple characters. If delimiter$ occurs between double quotes in string$ then it is ignored. This behavior can be changed by setting OPTION QUOTED to FALSE.
If the indicated position is outside a valid range, the original string is returned. Use the FLATTEN$ function to flatten out the returned token. See also EXCHANGE$, TOKEN$ and SPLIT.
Examples:

```
PRINT CHANGE$("a b c d \"e f\" g h i j", 5, "OK")
PRINT CHANGE$("a,b,c,d,e,f,g,h,i,j", 4, "123", ",")
```

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)
PRINT CHOP$("print \"end\"", "\n\r\t " & CHR$(34))
```

CHR$

CHR$(x)
Type: function
Returns the character belonging to ASCII number x. This function does the opposite of ASC. The value for x must lie between 0 and 255. See UTF8$ for Unicode values. Example:

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

CIPHER$

CIPHER$(connection)
Type: function
Returns the description of the encryption methods in use by the current network connection. Assumes that TLS has been enabled. See the chapter on secure network connections for more details. See also CA$ and CN$.

CL$

CL$
Type: variable
The Clear Line variable clears the line indicated by the current cursor position. See also EL$.

CLASS

CLASS
  <body>
ENDCLASS | END CLASS
Type: statement
Defines a class in original C++ code. This code is put unmodified into the generated global header source file. It is particularly useful when embedding C++ code into BaCon. See also USEC and USEH. Example:

```
CLASS TVision : public Tapplication
public:
    TVision() : TProgInit(&TVision::TV_initStatusLine,
&TVision::TV_initMenuBar, &Tvision::initDeskTop)
    {
    }
    static TMenuBar *TV_initMenuBar(TRect);
    static TStatusLine *TV_initStatusLine(TRect);
ENDCLASS
```

CLEAR

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

CLOSE

CLOSE FILE|DIRECTORY|NETWORK|SERVER|MEMORY|LIBRARY|DEVICE x[, y, z, ...]
Type: statement
Close file, directory, network, memory or library identified by handle. Multiple handles of the same type maybe used in a comma separated list. Examples:

```
CLOSE FILE myfile
CLOSE MEMORY mem1, mem2, block
CLOSE LIBRARY "libgtk.so"
```

CMDLINE

CMDLINE(options$)
Type: function
Defines the possible command line options to the current program. The CMDLINE function returns the ASCII value of each option until all provided options are parsed, in which case a '-1' is returned. In case an unknown option is encountered, question mark is returned.
If <options$> contains a colon, then an extra argument to the option is required. Such argument will appear in the reserved variable ARGUMENT$. Example where a program recognizes the options '-n' and '-f <arg>':

```
REPEAT
    option = CMDLINE("nf:")
    PRINT option
    PRINT ARGUMENT$
UNTIL option = -1
```

CN$

CN$(connection)
Type: function
Returns the Common Name from the certificate used in the current network connection. Assumes that TLS has been enabled. See the chapter on secure network connections for more details. See also CA$.

COIL$

**COIL$**([variable,] nr, expression$[, delimiter$])
Type: function
This is an inline loop which returns the results of the repeatedly evaluated <expression$> *appended* to a delimited string. The optional <variable> must be a numeric variable and <nr> defines how many times the <expression$> is carried out. If <variable> is not present then the anonymous variable "_" will be used.
It is possible to specify the concatenation delimiter explicitly. Note that this only works when COIL$ also defines a variable name, so COIL$ counts 4 parameters in total.

It is not allowed to use nested COIL$ constructs, or to use COIL$ multitple times in the same statement. Such code will generate a syntax error during conversion time. See also EXPLODE$ for another method to create delimited strings, the inline LOOP$ to create regular strings, or the inline if IIF$. More info on delimited strings can be found in the chapter on delimited string functions.
Example to get the first 10 letters in the Latin alphabet as a delimited string, using the anonymous variable:

```
PRINT COIL$(10, CHR$(64+_))
```

The following code prints only the even numbers between 0 and 100:

```
PRINT COIL$(100, IIF$(EVEN(_), STR$(_)))
```

This example prints the elements in a string array, each on a separate line:

```
PRINT COIL$(i, 5, array$[i-1], NL$)
```

COLLAPSE$

**COLLAPSE$**(string$ [,delim$])
Type: function
Collapses a delimited string so no empty items are present. The resulting string will only contain items separated by exactly one delimiter.
If the delimiter occurs between double quotes in string$, then it is ignored. This behavior can be changed by setting OPTION QUOTED to FALSE. See also the chapter on delimited string functions for more information about delimited strings. Example:

```
PRINT COLLAPSE$("a,,,b,,,,,,c,d,,,e", ",")
```

COLLECT

COLLECT <tree> **TO** <array> [**SIZE** <variable>][**STATIC**]
Type: statement
Retrieves all values of the nodes which are present in a binary tree. 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, COLLECT will declare the result array dynamically.
If COLLECT 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 the optional <variable>. This variable can be declared explicitly using LOCAL or GLOBAL.
For more information and examples, see the chapter on binary trees. See also FIND to verify the presence of a node in a binary tree. Example:

```
COLLECT mytree TO allnodes
FOR x = 0 TO UBOUND(allnodes)-1
    PRINT allnodes[x]
NEXT
```

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

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. The INTENSE and NORMAL keywords can be used combined with a color. This statement only works well with ANSI compliant terminals. Refer to the TYPE statement to set the terminal font type. Example:

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

Instead of color names, it is also possible to use a numeric reference. The FG and BG indicators have an enumeration as well:

Item	Number
Black	0
Red	1
Green	2
Yellow	3
Blue	4
Magenta	5
Cyan	6
White	7
BackGround	0
ForeGround	1

Example using numeric color references:

```
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 deprecated but it still is available for compatibility reasons. It can be used for high performance string concatenation but care should be taken when using non-BaCon functions as arguments.
Instead, BaCon uses either the '&' symbol or the '+' symbol as infix string concatenation operator. These operators are more versatile and allow any kind of concatenation. The following is the same example using '&':

```
txt$ = "Help this is " & name$ & " carrying a strange " & thing$
```

This will work as well:

```
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, REPEAT/UNTIL or DOTIMES/DONE.
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 <from> **TO** <new> [**SIZE** length]
Type: statement
If <from> and <to> contain string values, then COPY copies a file to a new file. Example:

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

If the SIZE keyword is present, then COPY assumes a memory copy. Example copying one array to another:

```
OPTION MEMTYPE long
DECLARE array[5], copy[5] TYPE long
array[0] = 15
array[1] = 24
array[2] = 33
```

```
array[3] = 42
array[4] = 51
COPY array TO copy SIZE 5
```

COS

COS(x)
Type: function
Returns the calculated COSINE of x, where x is a value in radians. Example:

```
PRINT COS(RAD(45))
```

COUNT

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

```
PRINT COUNT("Hello world", ASC("l"))
OPTION UTF8 TRUE
PRINT COUNT(FILL$(5, 0x1F600), 0x1F600)
```

See also FILL$.

CR$

CR$
Type: variable
Represents the Carriage Return as a string.

CURDIR$

CURDIR$
Type: function
Returns the full path of the current working directory. See also ME$ or REALPATH$.

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"
```

CUT$

**CUT$**(string$, start, end [, delimiter$])
Type: function
Retrieves elements from a delimited string$ split by delimiter$, starting at <start> until <end> inclusive. The delimiter$ is optional. If it is omitted, then the definition from OPTION DELIM is assumed. When specified, it may consist of multiple characters.
If the <start> parameter is higher than <end>, the result will be the same as when the parameters were reversed.
If delimiter$ occurs between double quotes in string$, then it is ignored. This behavior can be changed by setting OPTION QUOTED to FALSE. See also HEAD$ and TAIL$, and the chapter on delimited string functions for more information about delimited strings. Example:

```
PRINT "Excerpt: ", CUT$("Rome Amsterdam Kiev Bern Paris London",
2, 4)
```

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 [,flag])
Type: function
Calculates the decimal value of x, where x should be passed as a string. The optional [flag] parameter determines the base to convert from. If flag = 0 (default) then base is hexadecimal. If flag lies between 2 and 36 then the corresponding base is assumed. Note that DEC always returns a positive number. See also HEX$ and BIN$ for hexadecimal and binary conversions. Example:

```
PRINT DEC("AB1E")
PRINT DEC("00010101", 2)
```

DECLARE

DECLARE <var>[,var2,var3,...] **TYPE | ASSOC | TREE** <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")
```

DEG

DEG(x)
Type: function
Returns the degree value of x radians. Example:

```
PRINT DEG(PI)
```

DEL$

**DEL$**(string$, pos [, delimiter$])
Type: function
Deletes a member at position <pos> from a delimited string$ split by delimiter$. The delimiter$ is optional. If it is omitted, then the definition from OPTION DELIM is assumed. When specified, it may consist of multiple characters.
If the <pos> parameter is smaller than 1 or bigger than the amount of members in <string$>, then the original string$ is returned.
If delimiter$ occurs between double quotes in string$, then it is ignored. This behavior can be changed by setting OPTION QUOTED to FALSE. See APPEND$ for adding members, and the chapter on delimited string functions for more information about delimited strings. Example:

```
PRINT DEL$("Rome Amsterdam Kiev Bern Paris London", 2)
```

DELETE

DELETE <**FILE**|**DIRECTORY**|**RECURSIVE**> <x$> [**FROM**] <binary tree>
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. The FROM keyword is used when deleting a node from a binary 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"
DELETE value FROM tree
```

DELIM$

**DELIM$**(string$, old$, new$)
Type: function
Changes the delimiter in string$ from old$ to new$. The new delimiter can be of different size compared to the old delimiter.
If the old delimiter occurs between double quotes in string$, then it is ignored. This behavior can be changed by setting OPTION QUOTED to FALSE. See also the chapter on delimited string functions for more information about delimited strings. Example:

```
PRINT "Changed delimiter: ", DELIM$("f,q,a,c,i,b,r,t,e,d,z,", ",",
"@@@")
```

DIRNAME$

**DIRNAME$**(filename$)
Type: function
Returns the pathname part of a given filename. See also REALPATH$ and BASENAME$.

DLE$

DLE$
Type: variable
The DOS Line Ending variable returns a carriage return and a newline character. See also NL$ and CR$.

DO

DO
  <body>
DONE
Type: statement
With DO/DONE a body of statements can be grouped together. This is useful in case of special compiler constructs like pragmas. Example:

```
PRAGMA omp parallel sections
DO
    <...code...>
DONE
```

DOTIMES

DOTIMES x
  <body>
  [**BREAK**]|[**CONTINUE**]
DONE
Type: statement
With DOTIMES/DONE a body of statements can be repeated for a fixed amount of times without the need for a variable. This is known as an anonymous loop. As with other loops, it can be prematurely exited by using BREAK. Also, part of the body can be skipped by the use of the CONTINUE statement. See FOR/NEXT, WHILE/WEND and REPEAT/UNTIL for setting up other types of loops. To refer to the current loop, the anonymous variable "_" can be used.
Example:

```
DOTIMES 10
    PRINT "This is loop number ", _, " in DOTIMES."
DONE
```

EDITBOM$

**EDITBOM$**(string$, action)
Type: function
Adds or deletes the UTF8 byte order mark from <string$>. If <action> is ‘0’ then the byte order mark will be removed from <string$>, if found in the string. If <action> is ‘1’ then the UTF8 byte order will be added to the string if it is not there already. See also HASBOM to determine if a string has a UTF8 byte order mark.

EL$

EL$
Type: variable
The Erase Line variable clears the line from the current cursor position towards the end of the line. See also CL$.

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.

ESCAPE$

**ESCAPE$**(string$)
Type: function
Parses the text in <string$> and converts special characters (like newline or Unicode) into their escaped version. This functionality mainly is used by C files or JSON files. The special characters are converted into default C escape sequences like '\n', '\r', '\t' etc, and into '\u' and '\U' for Unicode characters. Non-printable binary data in the string is not converted. See also UNESCAPE$ to do the opposite. Example:

```
PRINT ESCAPE$("Hello world   ")
```

EVAL

EVAL(x$)
Type: function
Returns the evaluated result of a mathematical function described in a string. This function relies on the presence of the 'libmatheval' library and development header files on the compiling platform.
The EVAL function first needs to be enabled using OPTION EVAL.
The syntax for the mathematical function follows the regular C syntax. This means that the

operators + (add), - (subtract), * (multiply), / (divide) and ^ (exponent) work as usual. More over, the following functions are supported as well: exp, log, sqrt, sin, cos, tan, cot, sec, csc, asin, acos, atan, acot, asec, acsc, sinh, cosh, tanh, coth, sech, csch, asin), acosh, atanh, acoth, asech, acsch, log2e, e, log10e, ln2, ln10, pi, pi_2, pi_4, 1_pi, 2_pi, 2_sqrtpi, sqrt, sqrt1_2 and abs.
The string may contain real variable names from the BaCon program. These will be evaluated automatically. The variables must be declared as floating type (double) before. Example:

```
OPTION EVAL TRUE
DECLARE x, y TYPE FLOATING
x = 3
y = 4
nr = 5
PRINT EVAL("x*x + y +" & STR$(nr) & " + 6")
```

EVEN

EVEN(x)
Type: function
Returns 1 if x is even, else returns 0.

EXCHANGE$

**EXCHANGE$**(haystack$, pos1, pos2 [, delimiter$])
Type: function
Exchanges the token at pos1 with the token at pos2 in haystack$ split by delimiter$. The delimiter$ is optional. If it is omitted, then the definition from OPTION DELIM is assumed. When specified, it may consist of multiple characters. If delimiter$ occurs between double quotes in haystack$ then it is ignored. This behavior can be changed by setting OPTION QUOTED to FALSE.
If one of the indicated positions is outside a valid range, the original string is returned. Use the FLATTEN$ function to flatten out the returned token. See also CHANGE$, TOKEN$ and SPLIT.
Examples:

```
PRINT EXCHANGE$("a b c d \"e f\" g h i j", 8, 5)
PRINT EXCHANGE$("a,b,c,d,e,f,g,h,i,j", 4, 7, ",")
```

The next example code snippet sorts a delimited string. It uses the Bubble Sort algorithm:

```
t$ = "Kiev Amsterdam Lima Moscow Warschau Vienna Paris Madrid Bonn
Bern Rome"
total = AMOUNT(t$)
WHILE total > 1
    FOR x = 1 TO total-1
        IF TOKEN$(t$, x) > TOKEN$(t$, x+1) THEN t$ = EXCHANGE$(t$,
x, x+1)
    NEXT
    DECR total
WEND
```

Note that this is just an example to demonstrate the EXCHANGE$ function. Delimited strings can be sorted using the native SORT$ function.

EXEC$

**EXEC$**(command$ [, stdin$[, out]])
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. Also optionally, a third argument can be specified to determine whether all output needs to be captured (0 = default), only stdout (1) or only stderr (2). See SYSTEM to plainly execute a system command. Example:

```
result$ = EXEC$("ls -l")
result$ = EXEC$("bc", "123*456" & NL$ & "quit")
PRINT EXEC$("ls xyz123", NULL, 2)
```

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.

EXPLODE$

**EXPLODE$**(string$, [length [, delimiter$]])
Type: function
Splits a string based on <length> characters and returns the result in a delimited string using the default delimiter. The <length> parameter is optional. If not specified then the default value is 1. Also the delimiter$ is optional. If it is omitted, then the definition from OPTION DELIM is assumed. When specified, it may consist of multiple characters.
See also SPLIT to create an array based on a delimiter, MERGE$ to join the components of a delimited string back to a regular string, and the chapter on delimited string functions for more information about delimited strings. Example:

```
PRINT EXPLODE$("Amsterdam", 2)
```

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 where OPTION COMPARE establishes case insensitive expression matching. See also REPLACE$.

Examples:

```
PRINT EXTRACT$("bacon program", "ra")
PRINT EXTRACT$(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. This function also can be used to verify if a directory exists. If <filename$> is a symbolic link, it is dereferenced.

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 character <y>. The value for y must lie between 0 and 127 in ASCII mode, or between 0 and 1114111 (0x10FFFF) in case OPTION UTF8 is enabled. Example printing 10 times the character '@':

```
PRINT FILL$(10, ASC("@"))
```

Example printing 5 times a smiley character using unicode:

```
OPTION UTF8 TRUE
PRINT FILL$(5, 0x1F600)
```

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

FIND

FIND(binary tree, value)
Type: function
Verifies the presence of a value in a binary tree. For more information and examples, see the chapter on binary trees.

FIRST$

**FIRST$**(string$, amount [, delimiter$])
Type: function
Retrieves the remaining elements except the last <amount> from a delimited <string$> split by delimiter$. The delimiter$ is optional. If it is omitted, then the definition from OPTION DELIM is assumed. When specified, it may consist of multiple characters.
If no delimiter is present in the string, this function will return the full <string$>.
If delimiter$ occurs between double quotes in string$, then it is ignored. This behavior can be changed by setting OPTION QUOTED to FALSE.
See also TAIL$ to obtain elements counting from the end from the delimited string, and HEAD$ to obtain elements counting from the start. Refer to the chapter on delimited string functions for more

information about delimited strings. Example:

```
PRINT "Remaining first members: ", FIRST$("Rome Amsterdam Kiev
Bern Paris London", 2)
```

FLATTEN$

**FLATTEN$**(txt$ [, groupingchar$])
Type: function
Flattens out a string where the double quote symbol is used to group parts of the string together. Instead of the double quote symbol a different character can be specified (optional). See also UNFLATTEN$ for the reverse operation. Examples:

```
PRINT FLATTEN$("\"Hello \\\"cruel\\\" world\"")
PRINT FLATTEN$(TOKEN$("Madrid,Kiev,\"New York\",Paris", 3, ","))
PRINT FLATTEN$("\'Hello world\'", "'")
```

FLOOR

FLOOR(x)
Type: function
Returns the rounded down value of x. Note that this function returns a float value. Refer to CEIL for rounding up.

FOR

FOR var = x **TO**|**DOWNTO** y [**STEP** z]
  <body>
  [**BREAK**]|[**CONTINUE**]
NEXT [var]
FOR var$ **IN** source$ [**STEP** delimiter$]
  <body>
  [**BREAK**]|[**CONTINUE**]
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 (to) or decreased (downto) 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 of FOR, the variable <var$> will be assigned the space delimited strings mentioned in source$. Instead of a space delimiter, some other delimiter can be specified after the STEP keyword. The delimiter can consist of multiple characters. If the <delimiter$> occurs in between double quotes, then it is skipped until FOR finds the next one. This behavior can be changed by setting OPTION QUOTED to FALSE.
To prevent empty results, OPTION COLLAPSE can be set to TRUE (or 1). See also SPLIT to create an array of delimited strings.

Example:

```
OPTION COLLAPSE TRUE
FOR x$ IN "Hello cruel world"
    PRINT x$
NEXT
FOR y$ IN "1,2,\"3,4\",5" STEP ","
    PRINT y$
NEXT
```

FORK

FORK
<child>
[**ENDFORK** [x]] | [**END FORK** [x]]
Type: function
Duplicate the current running program in memory. If the return value is 0, then we're in the child process. If the child process needs an explicit exit then ENDFORK can be used.
If the return value > 0, then we are in the parent process; the actual value is the process ID of the spawned child.
If the return value < 0, then an error has occurred. See also REAP to detect and cleanup child processes which have ended, or SIGNAL to prevent occurring zombie processes altogether.
Example:

```
pid = FORK
IF pid = 0 THEN
    PRINT "I am the child, my PID is:", MYPID
    ENDFORK
ELIF pid > 0 THEN
    PRINT "I am the parent, pid of child:", pid
    REPEAT
        PRINT "Waiting for child to exit"
        SLEEP 50
    UNTIL REAP(pid)
ELSE
    PRINT "Error in fork"
ENDIF
```

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[, y, z, ...]
Type: statement
Releases claimed memory (see also MEMORY). Multiple memory pointers can be provided.
Example:

```
mem1 = MEMORY(500)
mem2 = MEMORY(100)
FREE mem1, mem2
```

This statement also can be used to delete individual members from associative arrays:

```
FREE array$("abc")
```

Lastly, it can delete all members of an associative array in one step:

```
FREE array$
```

FUNCTION

FUNCTION <name> ()|(STRING s, NUMBER i, FLOATING f, VAR v SIZE t) [**TYPE** <c-type>]
  <body>
  RETURN <x>
ENDFUNC | **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. If no type is specified, then BaCon will recognize the argument type from the variable suffix. In case no suffix is available, plain NUMBER type is assumed. 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. Function names also may end with the '#' or '%' type suffix, to force a float or integer return type.
Furthermore, it is also possible to explicitly define the type of the return value using the TYPE keyword.
Examples:

```
FUNCTION fh2celsius(FLOATING fahrenheit) TYPE float
    PRINT "Calculating Celsius..."
    RETURN (fahrenheit-32)*5/9
ENDFUNCTION
FUNCTION Hello$(STRING name$)
    RETURN "Hello " & name$ & " !"
ENDFUNCTION
```

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> [**FTYPE** <var>]
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. Optionally, the FTYPE keyword can store the actual file type of the file. The resulting file type numbering follows the same schedule as the FILETYPE function. Note that this optional feature is only supported on a few file systems, like btrfs, ext2, ext3, ext4, recent xfs, etc. Example:

```
OPEN "/tmp" FOR DIRECTORY AS mydir
REPEAT
    GETFILE myfile$ FROM mydir FTYPE thetype
    PRINT "File found: '", myfile$, "' - type: ", thetype
UNTIL ISFALSE(LEN(myfile$))
CLOSE DIRECTORY mydir
```

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 NETWORK or OPEN FOR SERVER. Example in case of a client connection:

```
website$ = "www.basic-converter.org"
OPEN website$ & ":443" FOR NETWORK AS mynet
PRINT "Remote IP is: ", GETPEER$(mynet)
```

Example when setting up a TCP server:

```
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**|**TREE** <c-type> | [**ARRAY** <size>]
Type: statement
Explicitly declares a variable to a C-type. The ASSOC keyword is used to declare associative arrays. The TREE keyword is used to declare binary trees. 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
```

GUIDEFINE

GUIDEFINE(string$)
Type: function
Defines a graphical user interface according to an object/property model. Each object (e.g. widget) should occur between curly brackets and should contain properties known to the used toolkit. By default, BaCon will assume the Xaw toolkit, but this can be overridden by the PRAGMA GUI statement. The GUIDEFINE function can only occur once in a BaCon program. The return value is a unique handle which should be used in subsequent GUI functions. For each object, the 'type and 'name' fields are obligatory. See also the chapter on GUI programming for more information. For example:

```
id = GUIDEFINE("{ type=window name=window
resources=\"*font:lucidasans-18\" XtNtitle=\"Hello world
application\" }"
```

GUIEVENT$

GUIEVENT$(handle [,TRUE])
Type: function
Executes the mainloop of the GUI and returns the name of the widget or string from the callback definition. For example:

```
WHILE TRUE
    SELECT GUIEVENT$(gui)
        CASE "window"
            BREAK
        CASE "button"
            INCR clicked
    ENDSELECT
WEND
```

The optional second argument allows returning a pointer to data which was passed to the internal callback function.

GUIFN

GUIFN(id, name$, function [, argn])
Type: function
Calls a GUI helper function based on a previously defined function pointer. It allows to omit specific type casting and does not perform argument type checks allowing more code flexibility. For example:

```
LOCAL (*show)() = XtPopup TYPE void
CALL GUIFN(id, "window", show, XtGrabNonexclusive)
```

GUIGET

GUIGET(id, name$, property$, &variable)
Type: function

Fetches a value set by <property$> from a widget with <name$> in a GUI identified by <id>. The <variable> should be a pointer variable, also in case of strings. For example:

```
CALL GUIGET(dialog, "dialog", XtNvalue, &data)
```

GUISET

GUISET(id, name$, property$, variable)
Type: function
Sets a value for <property$> regarding widget with <name$> in a GUI identified by <id>. For example:

```
CALL GUISET(dialog, "label", XtNjustify, XtJustifyLeft)
```

GUIWIDGET

GUIWIDGET(name$)
Type: function
Returns the memory address of a widget based on <name$>. This can come handy when using foreign GUI functions natively.

HASBOM

HASBOM(string$)
Type: function
Determines whether the string contains a UTF8 byte order mark. Some UTF8 text files are shipped with a first three bytes of 0xEF 0xBB 0xBF. The byte order mark is an optional byte sequence to indicate that the text contains UTF8 encoding. If <string$> contains such byte order mark HASBOM will return TRUE (1). Otherwise it will return FALSE (0). See also EDITBOM$ and the chapter on UTF8 encoding.

HASDELIM

HASDELIM(string$ [,delimiter$])
Type: function
Determines whether the string contains a delimiter or not. If the delimiter is not present, FALSE (0) is returned. Otherwise this function returns the position of the first delimiter in the string.
The delimiter$ is optional. If it is omitted, then the definition from OPTION DELIM is assumed. When specified, it may consist of multiple characters. If delimiter$ occurs between double quotes in string$ then it is ignored. Refer to the chapter on delimited string functions for more information about delimited strings.
For regular strings, see INSTR.

HASH

HASH(data [, length])
Type: function
Returns a hash from string data or memory data. The optional <length> specifies the amount of

bytes to be hashed. If no length is specified, the HASH function will use each byte until a 0 is encountered (like in C strings). On 64bit systems the returned value is 64bit, on 32bit systems the returned value is 32bit.
The hash algorithm used is based on the Fowler-Noll-Vo algorithm (FNV1a). Example to create a hash from a string:

```
PRINT HASH("Hello world") FORMAT "%lu\n"
```

Example to create a hash from data in memory:

```
mem = MEMORY(16)
FOR x = 0 TO 15
    POKE mem+x, RANDOM(256)
NEXT
PRINT HASH(mem, 16) FORMAT "%lu\n"
```

HEAD$

**HEAD$**(string$, amount [, delimiter$])
Type: function
Retrieves the first <amount> elements from a delimited string$ split by delimiter$. The delimiter$ is optional. If it is omitted, then the definition from OPTION DELIM is assumed. When specified, it may consist of multiple characters.
If delimiter$ occurs between double quotes in string$, then it is ignored. This behavior can be changed by setting OPTION QUOTED to FALSE.
See also LAST$ to obtain the remaining elements, and TAIL$ to obtain elements counting from the end of the delimited string. Refer to the chapter on delimited string functions for more information about delimited strings. Example:

```
PRINT "First 2 members: ", HEAD$("Rome Amsterdam Kiev Bern Paris
London", 2)
```

HEX$

HEX$(x)
Type: function
Calculates the hexadecimal value of x. Returns a string with the result. See also DEC to convert back to decimal.

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")
```

HOSTNAME$

HOSTNAME$()
Type: function
Retrieves the actual hostname of the current system where the program is running. Example:

```
PRINT "My hostname is: ", HOSTNAME$
```

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
```

The IF statement also allows comparing strings. The textual order is determined by the standard ASCII table. As a result, capital letters, which occur in the ASCII table before the small letters, are considered to be 'smaller' than regular letters.

```
name$ = "BaCon"
IF name$ > "basic" THEN
    PRINT "Not printed"
ELSE
    PRINT "This is correct!"
END IF
```

Equations allow the BETWEEN operator both for numbers and strings:

```
IF "c" BETWEEN "basic"; "pascal" THEN PRINT "This is C"
```

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
```
It is not allowed to mix an IF without a body and an ELSE which contains a body, or v.v. For example, the following is not allowed:
```
IF year > 1969 THEN PRINT "You are younger"
ELSE
    PRINT "You are older"
ENDIF
```

IIF / IIF$

IIF(expression, true[, false])
IIF$(expression, true[, false])
Type: function
The inline IF behaves similar to a regular IF, except that it is used as a function. The first argument contains the expression to be evaluated, the second argument will be returned when the expression is true, and the optional last argument will be returned when the expression is false.
The inline IF function also allows comparing strings. The textual order is determined by the standard ASCII table. As a result, capital letters, which occur in the ASCII table before the small letters, are considered to be 'smaller' than regular letters.
If the returned values are numeric, a plain IIF must be used. If strings are returned, then IIF$ should be used. See also LOOP$ for inline loops. Examples:
```
nr = IIF(1 <> 2, 10, 20)
answer$ = IIF$(2 + 2 = 5, "Correct", "Wrong")
PRINT IIF$(a$ = "B", "Yes it is")
PRINT IIF(x BETWEEN y;z, 1, -1)
```

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 return value. 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.
When the library name is 'NULL', a function will be imported from the program itself. In such situation, the ALIAS keyword is obligatory. Note that the program must be compiled with a linker flag like '-export-dynamic' (GCC) or '-rdynamic' (TCC) to make the target function visible for IMPORT.
An imported library can also be closed afterwards by using CLOSE LIBRARY. This will unload any symbols from the current program and release the library. It is mandatory to close a library when symbols need to be reloaded starting from the first IMPORT statement.
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"
IMPORT "MyFunc(void)" FROM NULL TYPE int ALIAS "Othername"
CLOSE LIBRARY "libm.so"
```

INBETWEEN$

**INBETWEEN$**(haystack$, lm$, rm$ [,flag])
Type: function
This function returns a substring from <haystack$>, delimited by <lm$> on the left and <rm$> on the right. The delimiters may contain multiple characters. They are not part of the returned result. The <flag> is optional (default value is 0) and specifies if <rm$> should either indicate the most right match (greedy indication, flag=1), or should return a balanced match (flag=2). See also OUTBETWEEN$. Note that OPTION COLLAPSE has no impact on this function. Example usage:

```
PRINT INBETWEEN$("Lorem ipsum dolor sit amet", "ipsum", "sit")
PRINT INBETWEEN$("<p>Chapter one.</p>", "<p>", "</p>")
a$ = INBETWEEN$("yes no 123 yes 456 yes", "no", "yes", TRUE)
```

INCLUDE

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

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

INCR

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

INDEX

INDEX(<array>, <value> [, flag])
Type: function
This function looks up a <value> in <array> and returns the position in the array where the value was found. If the value was not found, then this function returns 0.
By default, BaCon will perform a plain linear lookup. If the optional [flag] is set to TRUE, then BaCon assumes the array is ordered and the lookup will use the binary search algorithm, resulting in a better performance. This function only works for static and dynamic arrays. For associative arrays, refer to INDEX$. Examples:

```
DECLARE data[] = { 0, 1, 3, 5, 7, 8, 12, 30, 44, 55, 61 }
```

```
PRINT INDEX(data, 61, TRUE)
DECLARE floats#[] = { 3.2, 5.3, 7.5, 8.6, 12.2, 30.3, 55.5, 61.6}
PRINT INDEX(floats#, 30.3)
DECLARE string$[] = { "a", "b", "c", "d", "e" }
PRINT INDEX(string$, "c")
DECLARE nr TYPE int ARRAY 5
nr[0] = 1
nr[1] = 2
nr[2] = 6
nr[3] = 8
nr[4] = 10
PRINT INDEX(nr, 10)
```

INDEX$

INDEX$(<associative_array>, <value>)
Type: function
This function looks up a <value> in <associative_array> and returns the index in the array where the value was found. If the value was not found, then this function returns an empty string. If INDEX$ refers to a value which occurs multiple times, the returned index is the first inserted into the associative array. Example:

```
DECLARE data$ ASSOC STRING
data$("name") = "BaCon"
data$("place") = "Internet"
data$("language") = "BASIC"
PRINT INDEX$(data$, "Internet")
```

INPUT

INPUT [text[, ... ,]<variable[$]>
Type: statement
Interactive input from the user. If the variable ends with a '$' 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
```

The INPUT statement by nature is a blocking statement. Note that INPUT always will chop off a trailing newline from text captured into a string variable, if there is any.
The OPTION INPUT parameter can be used to define where INPUT should cut off the incoming stream from STDIN. This is especially useful in CGI programs, for example:

```
OPTION INPUT CHR$(4)
INPUT data$
```

INSERT$

**INSERT$**(source$, x, string$)
Type: function
Inserts the string$ into source$ at position <x>. The letters in source$ starting from position <x> are pushed forward. If x <= 1 then string$ is prepended to source$. If position > length of source$ then string$ is appended to source$. Example:

```
PRINT INSERT$("Hello world", 7, "cruel ")
```

INSTR

INSTR(haystack$, needle$ [,z])
Type: function
Returns the position where needle$ begins in haystack$, optionally starting at position z. If not found then this function returns the value '0'. See also TALLY to count the occurrences of needle$.

```
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.

INVERT

INVERT(<assoc_array>)
Type: function
Swaps the keys and values in an associative array. In case multiple keys hold the same value, a collision occurs. In such case, the last occurrence of a value is inverted and the INVERT function returns the amount of collisions. If all values are unique, then this function returns 0.
Note that values of numerical associative arrays will silently be converted to strings and that the key names will be converted back to the numerical type of the array. So if the keys contain regular strings then these will be converted to 0 in case of numerical arrays. Example:

```
DECLARE example ASSOC int
```

```
example("10") = 33
example("20") = 12
example("30") = 44
example("40") = 44
PRINT INVERT(example)
PRINT OBTAIN$(example)
```

ISASCII

ISASCII(string$)
Type: function
Returns TRUE (1) if <string$> only contains ASCII data. If not, FALSE (0) is returned. See also TOASCII$. Example:

```
PRINT ISASCII("hello world")
```

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. For static and dynamic arrays, refer to INDEX. See also NRKEYS to find the amount of keys in an associative array. Example:

```
DECLARE array ASSOC int
array("hello") = 25
array("world") = 30
array("compound", "key") = 40
PRINT ISKEY(array, "goodbye")
PRINT ISKEY(array, "world")
PRINT ISKEY(array, "compound", "key")
```

ISTOKEN

ISTOKEN(string$, token$ [, delimiter$])
Type: function
Verifies if the <token$> occurs in a delimited <string$>. The delimiter$ is optional. If it is omitted, then the definition from OPTION DELIM is assumed. When specified, it may consist of multiple characters.
If delimiter$ occurs between double quotes in string$, then it is ignored. This behavior can be changed by setting OPTION QUOTED to FALSE.
If token$ was found in string$, then this function returns the actual position of the token in the

delimited string, counting from the left. Otherwise it returns '0'. See also TOKEN$. Example:

```
string$ = "Kiev Amsterdam Lima Moscow Warschau Vienna Paris Madrid
Bonn Bern Rome"
PRINT ISTOKEN(string$, "Paris")
```

ISTRUE

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

ISUTF8

ISUTF8(string$)
Type: function
Returns TRUE (1) if <string$> is compliant with UTF8 encoding. If not, FALSE (0) is returned. Note that also random binary data can accidentally be compliant to UTF8 and that ASCII data always is compliant to UTF8. See also ISASCII. Example:

```
PRINT ISUTF8("Después mañana voy para mi casa")
```

JOIN

JOIN <array> [**BY** <sub>] **TO** <string> [**SIZE** <value>]
Type: statement
This statement can join elements of a one dimensional string array into a single string. The optional argument in BY defines the delimiter string in between the array elements. If BY is omitted, then no delimiter is put in between the concatenated array elements. The result is stored in the <string> argument mentioned by the TO keyword. Optionally, the total amount of array elements to be joined can be defined in SIZE. If SIZE is omitted then all elements in the array will be joined together. See also SPLIT to do the opposite. Example:

```
DECLARE name$[3]
name$[0] = "Hello"
name$[1] = "cruel"
name$[2] = "world"
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. A label may not contain spaces.
A label can appear amid existing DATA statements. In such case, RESTORE may refer to a label so READ will start reading data from that point. The label must be unique throughout the whole program because DATA statements are visible globally.

LAST$

**LAST$**(string$, amount [, delimiter$])
Type: function
Retrieves the remaining elements except the first <amount> from a delimited string$ split by delimiter$. The delimiter$ is optional. If it is omitted, then the definition from OPTION DELIM is assumed. When specified, it may consist of multiple characters.
If no delimiter is present in the string, this function will return the full <string$>.
If delimiter$ occurs between double quotes in string$, then it is ignored. This behavior can be changed by setting OPTION QUOTED to FALSE. See also HEAD$ to get the first element(s), and the chapter on delimited string functions for more information about delimited strings. Example:

```
PRINT "Remaining members: ", LAST$("Rome Amsterdam Kiev Bern Paris London", 1)
```

LCASE$

**LCASE$**(x$)
Type: function
Converts x$ to lowercase characters and returns the result. Example:

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

LEFT$

**LEFT$**(string$[, amount])
Type: function
Returns <amount> characters from the left of <string$>. The <amount> argument is optional. If omitted, then LEFT$ by default will return 1 character. See also RIGHT$ and MID$.

LEN

LEN(x$)
Type: function
Returns the length of ASCII string x$. If OPTION UTF8 is enabled, then the LEN function will return the length of UTF8 formatted strings correctly as well. See also ULEN.

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
```

LINENO

LINENO
Type: variable

Contains the current line number of the program. This variable mainly is used for debugging purposes.

LOAD$

**LOAD$**(filename$)
Type: function
Returns a string with the content of the specified text file in one step. See BLOAD for loading binary files in one step, and OPEN/WRITELN/READLN/CLOSE to read and write to a file using a filehandle. Example:

```
content$ = LOAD$("bacon.bac")
PRINT "Content of 'bacon.bac': ", content$
```

LOCAL

LOCAL <var>[,var2,var3,...] [**TYPE**]|**ASSOC**|**TREE** <c-type> | [**ARRAY** <size>] [**STATIC**]
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 to other functions, subroutines or records, nor to the main program.
If the TYPE keyword is omitted then variables are assumed to be of 'long' type. If TYPE is omitted and the variable name ends with a '$' then the variable will be a string.
The ARRAY keyword is used to define a dynamic array, which can be resized with REDIM at a later stage in the program. The optional STATIC keyword allows the array to be returned from a function.
Examples:

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

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**]|[**SORT**[**DOWN**]]
Type: statement

Retrieves all index names created in an associative array. The results are stored in <array$>. As it sometimes is unknown how many elements this resulting array will contain, the array should not be declared explicitly. Instead, LOOKUP will declare the result array dynamically.
If LOOKUP is being used in a function or sub, then <array> will have a local scope. Else <array> will be visible globally, and can be accessed within all functions and subs.
The optional SORT keyword will first sort the elements in the associative array before returning the corresponding indexes. A sort in descending order can be accomplished by adding the DOWN keyword. As with the SORT statement, string and numeric C-types are supported.
The total amount of elements created in this array is stored in the optional <variable>. This variable can be declared explicitly using LOCAL or GLOBAL. See also OBTAIN$ to store index names into a delimited string. 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.

LOOP

LOOP([variable,] nr, expression)
Type: function
This is an inline loop which returns the *sum* of the repeatedly evaluated <expression> as a numeric value. The optional <variable> must be a numeric variable and <nr> defines how many times the <expression> is carried out. If <variable> is not present then the anonymous variable "_" will be used. It is not allowed to use nested LOOP constructs or to use LOOP multitple times in the same statement. Such code will generate a syntax error during conversion time. See also FILL$, or COIL$ for an inline loop creating delimited strings, or the inline if IIF$.
Example to add all numbers mentioned in a string variable:

```
a$ = "123456789"
PRINT LOOP(LEN(a$), VAL(MID$(a$, _, 1)))
```

LOOP$

**LOOP$**([variable,] nr, expression$)
Type: function
This is an inline loop which returns the result of the repeatedly evaluated <expression$> as a *concatenated* regular string. The optional <variable> must be a numeric variable and <nr> defines how many times the <expression$> is carried out. If <variable> is not present then the anonymous variable "_" will be used. It is not allowed to use nested LOOP$ constructs or to use LOOP$ multitple times in the same statement. Such code will generate a syntax error during conversion time. See also FILL$, or COIL$ for an inline loop creating delimited strings, LOOP for numeric calculations, or the inline if IIF$.
Example to get all letters in the Latin alphabet:

```
PRINT LOOP$(i, 26, CHR$(96+i))
```

Example to print all elements in a delimited list using the anonymous variable "_":

```
PRINT LOOP$(AMOUNT(list$), TOKEN$(list$, _))
```

MAKEDIR

MAKEDIR <directory>
Type: statement
Creates an 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"
```

MAP

MAP <array1> [,array2, ...array<n>] **BY** <function> **TO** <array> [**SIZE** <const|variable>] [**STATIC**]
Type: statement
Performs a mapping of a function towards one or more arrays, storing the results in another array. All arrays shall have one dimension. The target array can be declared previously with the DECLARE or LOCAL statement. Using LOCAL in combination with the optional STATIC keyword, the array is created so it can be returned from a function. However, if there is no explicit previous declaration, then the MAP statement will declare the target array implicitly. The STATIC keyword can be used here as well.
When the target array is declared implicitly, the following logic applies: if MAP is being used in a function or sub, then the target <array> will have a local scope. Else <array> will be visible globally, and can be accessed within all functions and subs.
The <function> can either be defined by DEF FN, or it can point to a regular function. Only the function name should be provided, not the arguments. Note that the amount of arguments must be the same as the amount of arrays to which the <function> is mapped.
The SIZE argument is optional. When SIZE is not provided, MAP will apply the <function> to all elements in the given arrays. Example:

```
DEF FN addition(x, y) = x+y
MAP array1, array2 BY addition TO result SIZE 5
```

In this example, the first 5 elements of array1 and array2 are used for the 'addition' function. The results are stored in the array 'result'. See also SUM for adding members within the same array.
String arrays are supported as well:

```
word$ = "Hello world this is a program"
DEF FN func(x$) = LEN(x$)
SPLIT word$ TO letter$ SIZE total
MAP letter$ BY func TO new SIZE total
```

Here, each word is put into an array, after which the length of the individual words is being calculated. The results are then stored in another array.

MATCH

MATCH(string$, pattern$, [amount [,delimiter$]])
Type: function
Compares the delimited elements between <string$> and <pattern$>. By default, the amount of

elements to compare is determined by the amount of elements in <string$>. Optionally, the <amount> of elements to be compared can be provided explicitly. The value -1 means the default amount of elements in <string$>.
The <pattern$> can use the wildcards '?' and '*'. In case of '?' a single element will be matched. The '*' wildcard can be used to match multiple elements. To match an actual '?' or '*' in <string$> the wildcard symbol has to be escaped in <pattern$> or should be written between double quotes. The use of multiple wildcards in <pattern$> is allowed.
Both the <string$> and the <pattern$> should be delimited by the same delimiter$. The delimiter$ argument is optional. If it is omitted, then the definition from OPTION DELIM is assumed. When specified, it may consist of multiple characters.
If delimiter$ occurs between double quotes in string$, then it is ignored. This behavior can be changed by setting OPTION QUOTED to FALSE. See also the chapter on delimited string functions for more information about delimited strings. Examples:

```
IF MATCH("a b c d", "a b e f", 2) THEN PRINT "Partially the same"
IF MATCH("a b c d", "a b *") THEN PRINT "Matched"
IF NOT(MATCH("a b c d", "a b ? c")) THEN PRINT "Not matched"
IF MATCH("a b \"c d\" d", "a b * d") THEN PRINT "Matched"
IF MATCH("a b *d", "a b \\* d") THEN PRINT "Matched"
IF MATCH("a,b,c,d", "a,b,c,d", -1, ",") THEN PRINT "Matched"
```

MAX / MAX$

MAX(x, y)
**MAX$**(x$, y$)
Type: function
Returns the maximum value of two numbers or two strings. In case of strings, this function will follow the ASCII table to determine the 'maximum' string. This means that small letters, which occur in the ASCII table after capital letters, will have priority. Example:

```
PRINT MAX(3, PI)
PRINT MAX$("hello", "HELLO")
```

MAXNUM

MAXNUM(type)
Type: function
This function returns the maximum value possible for a certain type. Example:

```
PRINT MAXNUM(short)
PRINT MAXNUM(long) FORMAT "%ld\n"
```

MAXRANDOM

MAXRANDOM
Type: variable
Reserved variable which contains the maximum value RND can generate. The actual value may vary on different operating systems.

ME$

ME$
Type: function
Returns the full path of the current active program. See also CURDIR$ and REALPATH$.

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. The following example creates a memory area to store integers:

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

Effectively, this will provide a memory area of 100 times the length of an integer.

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>.

MERGE$

**MERGE$**(string$ [, delimiter$])
Type: function
Merges the components of a delimited string to a regular string. The delimiter$ is optional. If it is omitted, then the definition from OPTION DELIM is assumed. When specified, it may consist of multiple characters.
See also EXPLODE$ to create a delimited string, and the chapter on delimited string functions for more information about delimited strings. Example:

```
PRINT MERGE$("A m s t e r d a m")
```

MID$

**MID$**(x$, y, [z])
Type: function
Returns z characters starting at position y in x$. If y is a negative number, then start counting the position from the end of x$. The parameter 'z' is optional. When this parameter is 0, negative or omitted, then everything from position 'y' until the end of the string is returned. See also RIP$ for the complement of MID$. Example:

```
txt$ = "Hello cruel world"
PRINT MID$(txt$, 7, 5)
PRINT MID$(txt$, -11)
PRINT MID$(txt$, 12, -1)
```

MIN / MIN$

MIN(x, y)
**MIN$**(x$, y$)
Type: function
Returns the minimum value of two numbers or two strings. In case of strings, this function will follow the ASCII table to determine the 'minimum' string. This means that capital letters, which occur in the ASCII table before the small letters, will have priority. Example:

```
PRINT MIN(3, PI)
PRINT MIN$("hello", "HELLO")
```

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.

MYPID

MYPID
Type: function
Returns the process ID of the current running program.

NANOTIMER

NANOTIMER
Type: function
Keeps track of the amount of nanoseconds the current program is running. See also TIMER to measure milliseconds. Example:

```
SLEEP 100
PRINT "Slept ", NANOTIMER, " nanoseconds!"
```

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 New Line 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.

NRKEYS

NRKEYS(array)
Type: function
Returns the amount of index names (keys) in the associative <array>. See ISKEY to find out if a key exists in an associative array. Refer to UBOUND for other types of arrays. Example:

```
DECLARE array ASSOC int
array("hello") = 25
array("world") = 30
PRINT NRKEYS(array)
```

OBTAIN$

**OBTAIN$**(assoc$ [,delimiter$ [, sort]])
Type: function
Retrieves all index names from an associative array and returns a delimited string split by delimiter$. Multiple indexes in the same element are returned space separated. The delimiter$ is optional. If it is omitted, then the definition from OPTION DELIM is assumed. When specified, it may consist of multiple characters. When NULL, the function will take the default delimiter as defined by OPTION DELIM.
The optional <sort> argument will first sort the values of the associative array before retrieving the index names, where TRUE will sort ascending and FALSE will sort descending.
See the chapter on delimited string functions for more information about delimited strings. See also LOOKUP to store index names from an associative array into a regular array. Example:

```
PRINT OBTAIN$(AssocArray, ",")
PRINT OBTAIN$(mydata, NULL, TRUE)
```

ODD

ODD(x)
Type: Function
Returns 1 if x is odd, else returns 0.

ON

ON x <**GOTO** label1 [, label2[, label<x>]]>|<**CALL** func1 [, func2 [, funcx]]>
Type: statement
Jump to a label or function based on the value of x. When x is 1 then the first item is chosen, when x is 2 the second item and so on. When x has a higher value than the available labels this statement is ignored. Example:

```
ON x GOTO a, b
PRINT "No label found"
END
LABEL a
    PRINT "a"
    END
LABEL b
    PRINT "b"
    END
```

Other example using functions, adding brackets to the called function name is optional:

```
ON ISTOKEN(list$, item$) CALL setx, sety(), print_str(str$)
```

OPEN

OPEN <file|dir|address> **FOR READING**|**WRITING**|**APPENDING**|**READWRITE**|**DIRECTORY**|**NETWORK** [**FROM** address[:port]]|**SERVER**|**MEMORY**|**DEVICE 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$ = 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> | <**PARSE** x> | <**FRAMEWORK** x> | <**VARTYPE** x> | <**QUOTED** x> | <**DQ** x> | <**ESC** x> | <**UTF8** x> | <**DELIM** x> | <**BREAK** x> | <**EXPLICIT** x> | <**INPUT** x> | <**ERROR** x> | <**PROPER** x> | <**TLS** x> | <**EVAL** x> | <**NO_C_ESC** 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 in the IF and IIF/IIF$ statements and in the BETWEEN operator and also in regular expressions with REPLACE$, EXTRACT$, REGEX and WALK$ 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, GETBYTE, COPY, ROL and ROR. Default value is 'char' (1 byte). Any valid C type can be used here, for example 'float', 'unsigned int', 'long' etc.
- The COLLAPSE argument specifies if the results of the SPLIT and FOR..IN statements and of the delimited string functions 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.
- The PARSE argument defines if BaCon should allow non-BaCon code. It can be used to embed foreign functions from external C libraries. Use with care, as this option accepts any random piece of text. Errors only will popup during compile time, which may be hard to troubleshoot. The default value is 1.
- The FRAMEWORK option is used in case of linking to MacOSX frameworks like Cocoa. This option allows multiple frameworks separated by a comma. For example: PRAGMA FRAMEWORK COCOA.
- The VARTYPE option defines the default variable type in case of implicit declarations. The default value for this option is: long.
- The QUOTED argument defines whether text delimiters appearing between double quotes should be skipped (1) or not (0). The default is to skip delimiters between double quotes (1).

- The DQ argument defines the symbol for OPTION QUOTED. This is a numeric ASCII value between 0 and 255. Default value is 34 (double quotes).
- The ESC argument defines the escape symbol for OPTION DQ. This is a numeric ASCII value between 0 and 255. Default value is 92 (backslash).
- The UTF8 argument enables all BaCon string functions to process text in UTF8 format correctly. The default is to process text as ASCII (0).
- The DELIM argument defines the delimiter string when processing delimited strings. It always should be provided as a static string literal. The default value is a single space " ".
- The BREAK argument prevents and disables the use of BREAK statements in generated C code. The default is to allow BREAK statement (TRUE).
- The EXPLICIT argument enforces the declaration of all variables used in a program. The default value is 0 (FALSE), so no variable declaration is enforced.
- The INPUT argument defines where the stream of input characters from STDIN should be cut off. By default, INPUT returns when a newline is encountered. The default value is "\n".
- The ERROR argument sets whether or not a program will exit as soon a runtime error occurs. The default is TRUE (exit upon error). If set to FALSE, the program must take care of error handling by itself, and setting an error callback will be possible. See also the chapter on error catching for more information.
- The PROPER argument sets whether or not the PROPER$ function should leave the items in a delimited string intact or not. The default value is FALSE, causing the PROPER$ function to lowercase the remainder of each item.
- The TLS argument enables network connections to be TLS encapsulated. See the chapter on secure network connections for more details. The default value is FALSE.
- The PRIORITY argument defines the TLS priority string for GnuTLS. By default, the priority string is defined as "NORMAL:-VERS-TLS1.3:%COMPAT" disabling TLS 1.3.
- The NO_C_ESC argument can disable the effects of the C escape symbol "\" in string literals. This can come handy when printing ASCII art for example. Default value is FALSE (do not disable effects of escape symbol in text).
- The EVAL argument enables code generation for the EVAL function. It also adds linking flags for 'libmatheval' to the generated Makefile. The default value is FALSE.

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.

OUTBETWEEN$

**OUTBETWEEN$**(haystack$, lm$, rm$ [,flag])

Type: function
This function returns <haystack$> where the substring delimited by <lm$> on the left and <rm$> on the right is cut out. The delimiters may contain multiple characters. They are not part of the returned result. The <flag> is optional (default value is 0) and specifies if <rm$> should either indicate the most right match (greedy indication, flag=1), or should return a balanced match (flag=2). See also INSERT$ to insert a string and INBETWEEN$ to return the delimited substring. Note that OPTION COLLAPSE has no impact on this function. Example usage:

```
PRINT OUTBETWEEN$("Lorem ipsum dolor sit amet", "ipsum", "sit")
a$ = OUTBETWEEN$("yes no 123 yes 456 yes", "no", "yes", TRUE)
```

PARSE

PARSE <string$> **WITH** <pattern$> [**BY** <delim$>] **TO** <array$> [**SIZE** <size>] [**STATIC**]
Type: statement
This statement can parse a delimited <string$> using a delimited <pattern$> containing wildcards. The matched results are stored in <array$> mentioned by the TO keyword. As sometimes it cannot be known in advance how many elements this resulting array will contain, the array may not be declared before with LOCAL or GLOBAL. The optional BY argument determines the delimiter. If the BY keyword is omitted then the default definition from OPTION DELIM will be used.
If PARSE is being used in a function or sub, then <array$> will have a local scope. Else <array$> will be visible globally, and can be accessed within all functions and subs.
The total amount of elements created in this array is stored in <size>. This variable can be declared explicitly using LOCAL or GLOBAL. The SIZE keyword is optional and may be omitted.
If <delim$> occurs in between double quotes, then it is skipped. This behavior can be changed by setting OPTION QUOTED to FALSE. If a double quote needs to be present in the <string$>, it must be escaped properly.
The provided <pattern$> must contain a delimited string with wildcards. The wildcard symbol '?' will match one delimited item, and the wildcard symbol '*' matches one or more items in the delimited string. Each time a match is found it will be added to <array$>. Parsing is executed from left to right and stops as soon matching the pattern fails.
Example usage:

```
OPTION BASE 1
PARSE "a b c d e f" WITH "a ? c * f" TO array$
FOR i = 1 TO UBOUND(array$)
    PRINT array$[i]
NEXT
```

This example will return two array items, one containing a single element and the other containing two delimited elements. See also MATCH to compare delimited strings using wildcards and COLLAPSE$ to make sure items in a string are separated by one delimiter.

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: 3.14159265358979323846.

POKE

POKE <x>, <y> [**SIZE** range]
Type: statement
Stores a value <y> at memory address <x>. The optional SIZE keyword can be used to store the value <y> in a complete range of addresses starting from address <x>. Use PEEK to retrieve a value from a memory address. Use OPTION MEMTYPE to determine the actual size of the type to store. Examples:

```
OPTION MEMTYPE float
mem = MEMORY(SIZEOF(float))
POKE mem, 32.123
area = MEMORY(1024)
POKE area, 0 SIZE 1024
```

POW

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

PRAGMA

PRAGMA <**OPTIONS** x> | <**LDFLAGS** x> [TRUE] | <**COMPILER** x> | <**INCLUDE** x> | <**RE** x [**INCLUDE** y] [**LDFLAGS** z]> | <**GUI** x>
Type: statement
Instead of passing command line 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. When no valid option to PRAGMA is provided, BaCon will translate to the plain compiler directive '#pragma'. 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 --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
```
The LDFLAGS argument can specify whether the flags should occur *before* other flags using TRUE:
```
PRAGMA LDFLAGS -Wl,--no-as-needed TRUE
```
Example specifying a regular expression engine like PCRE (see the chapter on Regular Expressions for more details):
```
PRAGMA RE pcre INCLUDE <pcreposix.h> LDFLAGS -lpcreposix
```
Example using an OpenMP pragma definition:
```
PRAGMA omp parallel for private(x)
```
Example specifying a GTK backend for the GUI functions:
```
PRAGMA GUI gtk3
```

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. As with most BASICs, the PRINT statement may be abbreviated using the '?' symbol. A semicolon at the end of the line prevents printing a newline. For printing to stderr, see EPRINT. Examples:
```
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 has to 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" FORMAT "%s" TO hello$
PRINT mytime FORMAT "Time is now: %d" TO result$ 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
```

PROPER$

**PROPER$**(string$ [,delimiter$])
Type: function

Capitalizes the first letter of all elements in a delimited string split by delimiter$. By default, other letters are put to lowercase. This behavior can be altered by setting OPTION PROPER.
The delimiter$ is optional. If it is omitted, then the definition from OPTION DELIM is assumed. When specified, it may consist of multiple characters.
If delimiter$ occurs between double quotes in string$, then it is ignored. This behavior can be changed by setting OPTION QUOTED to FALSE. See the chapter on delimited string functions for more information about delimited strings. Example:

```
PRINT PROPER$("hEllO crUEl wOrLd")
```

PROTO

PROTO <function name>[,function name [, ...]] [**ALIAS** word] [**TYPE** c-type]
Type: statement
Defines a foreign function so it is accepted by the BaCon parser. Multiple function names may be mentioned, but these should be separated by a comma. Optionally, PROTO accepts an alias which can be used instead of the original function name. Also, PROTO can define a c-type to define the type of return value for a foreign function.
During compilation the BaCon program must explicitly be linked with an external library to resolve the function name. See also OPTION PARSE to allow foreign functions unconditionally.
Examples:

```
PROTO glClear, glClearColor, glEnable
PROTO "glutSolidTeapot" ALIAS "TeaPot"
PROTO "gtk_check_version(int,int,int)" TYPE char*
```

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. The reserved variable SP provides the total amount of elements currently on the stack. 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.

RAD

RAD(x)
Type: function
Returns the radian value of x degrees. Example:

```
PRINT RAD(45)
```

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
```

REALPATH$

**REALPATH$**(filename$)
Type: function
Returns the absolute full path and name of a given filename. Symbolic links are resolved as well as relative references like '../'. See also CURDIR$.

REAP

REAP(pid)
Type: function
After a forked process has ended, it can turn into a so-called 'zombie' process. This function can remove such process from the process list, using the process ID as an argument. When the value -1 is used as argument, REAP will remove any zombie child process.
The return value of REAP indicates the process ID of the process which was removed from the process list successfully. If the return value is 0, then no child process has finished yet, and no process ID has been removed. When the return value is -1, an error has occurred (a common mistake is providing a wrong process ID value).
This function does not pause and returns immediately. For an example, refer to FORK.

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. In case of TLS connections, it is recommended to use a multitude of 16k for the chunk size.
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$ = total$ & dat$
UNTIL ISFALSE(WAIT(mynet, 500))
CLOSE NETWORK mynet
```

RECORD

RECORD <var>[**ARRAY** <x>]
  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 main program, it automatically will be visible globally. 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. See the chapter on records for more details on the usage of records. 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. Note that this function does not support non-greedy matching. See the chapter on regular expressions to specify different regular expression engines for more flexibility. 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. Note that an error occurs when the target directory is on a different partition. Example:

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

REPEAT

REPEAT
  <body>
  [**BREAK**]|[**CONTINUE**]
UNTIL <equation>
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. With CONTINUE part of the body can be skipped. The BETWEEN operator is allowed in the equation. Example:

```
REPEAT
    C = GETKEY
UNTIL C = 27
```

REPLACE$

**REPLACE$**(haystack$, needle$, replacement$ [, flag])
Type: function
Substitutes a substring <needle$> in <haystack$> with <replacement$> and returns the result. The replacement does not necessarily need to be of the same size as the substring. With the optional flag set to 1 the <needle$> should be taken as a regular expression, and OPTION COMPARE impacts case insensitive matching. With the optional flag set to 2, REPLACE$ will behave as a translate, meaning that the characters in <needle$> will be replaced by the successive characters in <replacement$>. See also EXTRACT$.
Examples:

```
PRINT REPLACE$("Hello world", "l", "p")
PRINT REPLACE$("Some text", "me", "123")
PRINT REPLACE$("Goodbye <all>", "<.*>", "123", TRUE)
PRINT REPLACE$("abc123def", "[[:digit:]]", "x", 1)
PRINT REPLACE$("Hello world", "old", "pme", 2)
```

RESIZE

RESIZE <x> **TO** <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$.

REV$

**REV$**(string$ [,delimiter$])
Type: function
Puts all elements in a delimited string split by delimiter$ in reverse order. The delimiter$ is optional. If it is omitted, then the definition from OPTION DELIM is assumed. When specified, it may consist of multiple characters.
If delimiter$ occurs between double quotes in string$, then it is ignored. This behavior can be changed by setting OPTION QUOTED to FALSE. See also the chapter on delimited string functions for more information about delimited strings. Example:

```
PRINT "Reverted members: ", REV$("Rome Amsterdam Kiev Bern Paris London")
```

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$**(string$[, amount])
Type: function
Returns <amount> characters from the right of <string$>. The <amount> argument is optional. If omitted, then RIGHT$ by default will return 1 character. See also LEFT$ and MID$.

RIP$

**RIP$**(x$, y, [z])
Type: function
This function is the complement of MID$. It returns the characters which are left after position <y> in <x$> with optional length <z> is omitted. If y is a negative number, then start counting the position from the end of x$. The parameter 'z' is optional. When this parameter is 0, negative or left out, then everything from position 'y' until the end of the string is omitted. Example:

```
txt$ = "Hello cruel world"
PRINT RIP$(txt$, 7, 5)
PRINT RIP$(txt$, -11)
PRINT RIP$(txt$, 12, -1)
```

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
```

ROL

ROL(nr)
Type: function
This function performs a binary shift to the left. The highest bit will be recycled into bit 0. The total amount of bits in a value is determined by the MEMTYPE option.

```
OPTION MEMTYPE short
PRINT ROL(32768)
```

ROR

ROR(nr)
Type: function
This function performs a binary shift to the right. The lowest bit will be recycled into the highest, depending on the setting of the MEMTYPE option.

```
OPTION MEMTYPE int
PRINT ROR(1)
```

ROTATE$

**ROTATE$**(string$, step [,delimiter$])
Type: function
Rotates all elements in a delimited string split by delimiter$ <step> positions forward. In case the <step> parameter is a negative number, the rotation will be backwards. The delimiter$ is optional. If it is omitted, then the definition from OPTION DELIM is assumed. When specified, it may consist of multiple characters.
If delimiter$ occurs between double quotes in string$, then it is ignored. This behavior can be changed by setting OPTION QUOTED to FALSE. See also the chapter on delimited string functions for more information about delimited strings. Example:

```
PRINT ROTATE$("Rome Amsterdam Kiev Bern Paris London", 2)
```

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.

RUN

RUN <command$>
Type: statement
Executes an operating system command thereby transferring control. This effectively means that the current program is left permanently, the process ID is preserved and that this statement does not return to the BaCon program. Typically, the RUN statement is used at the end of a BaCon program. It can only execute one system command at a time.
The behavior of RUN differs from the SYSTEM statement, which can execute a set of compound commands in a shell and can query the exit status. See also EXEC$ and RUN$.
Example:

```
RUN "ls -l"
```

RUN$

**RUN$**(command$ [, stdin$[, out]])
Type: function
Executes an operating system command in a coprocess and returns the resulting output to the BaCon program. Because the coprocess PID is not a shell, but the PID of the executed command itself, this function cannot return a system command exit status, and can only execute one system command at a time. Optionally, a second argument may be used to feed to STDIN. Also optionally, a third argument can be specified to determine whether all output needs to be captured (0 = default), only stdout (1) or only stderr (2). See RUN and SYSTEM to plainly execute a system command.
Example:

```
result$ = RUN$("ps x")
PRINT RUN$("rev", "This is a string", 1)
PRINT RUN$("ls -z", NULL, 2)
```

SAVE

SAVE string$ **TO** filename$
Type: statement
Saves a string to disk in one step. If the file already exists it is overwritten. See BSAVE for saving binary files in one step, and OPEN/WRITELN/READLN/CLOSE to read and write to a file using a

filehandle. Example:

```
SAVE result$ TO "/tmp/data.txt"
SAVE "Hello", "world" TO file$
```

SCREEN

SCREEN <**SAVE**> | <**RESTORE**>
Type: statement
This statement can save the state of the current ASCII screen into memory so it can be restored at a later moment in time. It only works with ANSI compliant terminals. Example:

```
SCREEN SAVE
PRINT "Hello world"
SCREEN RESTORE
```

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 [,flag])
Type: function
Searches for <string> in file opened with <handle>. The search returns the byte 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 data is not found, then the value '-1' is returned.
Optionally, a third argument can be used to determine where to start the search and in which direction the search should take place. The following values are accepted:
0: start at the beginning of the file, search forward (default)
1: start at the current position of the filepointer, search forward
2: start at the current position of the filepointer, search backward
3: start at the end of the file, search backward.
Note that when searching backwards, the actual search begins at the start position minus the length of the searched string.

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. Note that in such situation, each CASE keyword will re-evaluate the expression at each occurrence.
Also, BaCon knows how to 'fall through' by either using a semicolon or a comma separated list, in case multiple values lead to the same result:

```
SELECT human$
    CASE "Man"
        PRINT "It's male"
    CASE "Woman", "Girl"
        PRINT "It's female"
    CASE "Child";
    CASE "Animal"
        PRINT "It's an 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
```

Example setting the terminal to raw input mode (no echo and no line based input):

```
SETSERIAL STDIN_FILENO LMODE ~ECHO
SETSERIAL STDIN_FILENO LMODE ~ICANON
```

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.

SIGNAL

SIGNAL <sub>, <signal>
Type: statement
This statement connects a Unix signal to a callback function. Plain POSIX signal names can be used, for example SIGINT, SIGTERM, SIGCHLD and so on. Next to that, this statement accepts the SIG_DFL (default action) and SIG_IGN (ignore signal) symbols for a callback also.
Example to ignore the SIGCHLD signal, preventing zombie processes to occur:

```
SIGNAL SIG_IGN, SIGCHLD
```

Example connecting the <CTRL>+<C> signal to a SUB:

```
SUB Cleanup                     : ' Signal callback function
    SIGNAL SIG_DFL, SIGINT      : ' Restore CTRL+C
    PRINT "Cleaning up"         : ' Do your cleanup here
    STOP SIGINT                 : ' Send the SIGINT to myself
ENDSUB
SIGNAL Cleanup, SIGINT          : ' Catch CTRL+C
PRINT "Waiting..."
key = GETKEY
```

SIN

SIN(x)
Type: function
Returns the calculated SINUS of x, where x is a value in radians.

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 be indexed or associative.
For indexed arrays, the amount of elements to sort can be specified with the optional keyword SIZE. Also optionally, the keyword DOWN can be used to sort in descending order.
For associative arrays, sorting means changing the insertion order of the key/value pairs in the hash table. The ordered keys can be retrieved by LOOKUP or OBTAIN$. The SIZE keyword has no impact.
Examples:

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

Sorting an associative array:

```
DECLARE aa ASSOC short
aa("one") = 33
aa("two") = 12
aa("three") = 44
aa("four") = 15
aa("five") = 8
SORT aa DOWN
```

SORT$

**SORT$**(string$ [,delimiter$])
Type: function
Sorts all elements in a delimited string split by delimiter$. The delimiter$ is optional. If it is omitted, then the definition from OPTION DELIM is assumed. When specified, it may consist of multiple characters.
If delimiter$ occurs between double quotes in string$, then it is ignored. This behavior can be changed by setting OPTION QUOTED to FALSE. See also the chapter on delimited string functions for more information about delimited strings. Example:

```
PRINT "Sorted members: ", SORT$("f,q,a,c,i,b,r,t,e,d,z,", ",")
```

SOURCE$

SOURCE$
Type: variable
Reserved variable which contains the BaCon source code of the current running program. Note that for commercial programs this variable should not be used, because it stores the source code as plain text in the resulting binary.

SP

SP
Type: variable
Reserved variable containing the amount of elements currently in the stack. See also PUSH and PULL.

SPC$

SPC$(x)
Type: function
Returns an x amount of spaces.

SPLIT

SPLIT <string$> [**BY** <substr$>|<nr>] **TO** <array$> [**SIZE** <variable>] [**STATIC**]
Type: statement
This statement can split a string into smaller pieces. The optional BY argument determines where the string is being split. If the BY keyword is omitted then the definition from OPTION DELIM is used to split string$. The results are stored in the argument <array$> mentioned by the TO keyword. As sometimes it cannot be known in advance how many elements this resulting array will contain, the array may not be declared before with LOCAL or GLOBAL.
If <substr$> occurs between double quotes in string$, then it is ignored. This behavior can be changed by setting OPTION QUOTED to FALSE. See the chapter on delimited string functions for more information about delimited strings.
If SPLIT is being used in a function or sub, then <array$> will have a local scope. Else <array$> will be visible globally, and can be accessed within all functions and subs.
The total amount of elements created in this array is stored in <variable>. This variable can be declared explicitly using LOCAL or GLOBAL. The SIZE keyword is optional and may be omitted.
If the <substr$> delimiter occurs in between double quotes, then it is skipped. This behavior can be changed by setting OPTION QUOTED to FALSE. If a double quote needs to be present in the <string$>, it must be escaped properly.
If the value for BY is numeric, then string$ will be cut in pieces each containing <nr> characters. If <nr> is 0 then there are no results. If <nr> is equal to or bigger than the length of the string, then the original string will be returned as an array with one element.
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.

```
SPLIT "one,two,\"three,four\",five" BY "," TO array$ SIZE dim
```

This will return 4 elements, because one separator (the comma) lies in between double quotes.

The optional STATIC keyword allows the created <array> to be returned from a function. See also EXPLODE$ to split text returning a delimited string, TOKEN$ to retrieve one single element from a delimited string, and JOIN to join array elements into a string.

SQR

SQR(x)
Type: function
Calculates the square root from a number.

STOP

STOP [signal]
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>. The STOP statement actually sends the 'STOP' signal to the current program. Optionally, a different signal can be defined. The signal can be a number or a predefined name from libc, like SIGQUIT, SIGKILL, SIGTERM and so on. Example sending the <CTRL>+<C> signal:

```
STOP SIGINT
```

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. If no type is specified, then BaCon will recognize the argument type from the variable suffix. In case no suffix is available, plain NUMBER type is assumed. 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
```

SUM / SUMF

SUM(array, amount [,minimum])
SUMF(array, amount [,minimum])
Type: function
Returns the sum of <amount> elements in <array>. Optionally, a check can be added which specifies the minimum value for each element to be added. If an array element falls below the specified value then it is excluded from the sum calculation.
The SUM and SUMF functions perform the same task, but SUM requires an array with integers and SUMF an array with floating values. See also MAP. Example:

```
PRINT SUM(ages, 10)
PRINT SUMF(temperatures, 100, 25.5)
```

SWAP

SWAP x, y
Type: statement
Swaps the contents of the variables x and y. The types of the variables can be mixed. Note that when swapping an integer with a float precision may be lost.
Numeric variables can be swapped with string variables, thereby effectively converting types.
Example:

```
SWAP x%, y#
SWAP number, string$
```

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.

TAIL$

**TAIL$**(string$, amount [, delimiter$])
Type: function
Retrieves the last <amount> elements from a delimited string$ split by delimiter$. The delimiter$ is optional. If it is omitted, then the definition from OPTION DELIM is assumed. When specified, it

may consist of multiple characters.
If delimiter$ occurs between double quotes in string$, then it is ignored. This behavior can be changed by setting OPTION QUOTED to FALSE.
See also FIRST$ to obtain the remaining elements from the delimited string, and HEAD$ to obtain elements counting from the start. Refer to the chapter on delimited string functions for more information about delimited strings. Example:

```
PRINT "Last 2 members: ", TAIL$("Rome Amsterdam Kiev Bern Paris London", 2)
```

TALLY

TALLY(haystack$, needle$ [,z])
Type: function
Returns the amount of times needle$ occurs in haystack$, optionally starting at position z. If the needle$ is not found, then this function returns the value '0'. See INSTR to find the position of a string. Example:

```
amount = TALLY("Hello world are we all happy?", "ll")
PRINT TALLY("Don't take my ticket", "t", 10)
```

TAN

TAN(x)
Type: function
Returns the calculated tangent of x, where x is a value in radians.

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. See NANOTIMER to measure nanoseconds. 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)
```

TOASCII$

**TOASCII$**(string$)
Type: function
Returns the same string of which each byte has bit 7 set to 0. Note that this can lead to unpredictable results. See also ISASCII or ISUTF8. Example:

```
PRINT TOASCII$("Hello world")
```

TOKEN$

**TOKEN$**(haystack$, n [, delimiter$])
Type: function
Returns the n^{th} token in haystack$ split by delimiter$. The delimiter$ is optional. If it is omitted, then the definition from OPTION DELIM is assumed. When specified, it may consist of multiple characters.
If delimiter$ occurs between double quotes in haystack$, then it is ignored. This behavior can be changed by setting OPTION QUOTED to FALSE.
If the indicated position is outside a valid range, the result will be an empty string. Use the FLATTEN$ function to flatten out the returned token. See also ISTOKEN, AMOUNT and SPLIT.
Examples:

```
PRINT TOKEN$("a b c d \"e f\" g h i j", 6)
PRINT TOKEN$("Dog Cat @@@ Mouse Bird@@@ 123@@@", 3, "@@@")
PRINT TOKEN$("1,2,3,4,5", 3, ",")
PRINT TOKEN$("1,2," & CHR$(34) & "3,4" & CHR$(34) & ",5,", 6, ",")
```

TOTAL

TOTAL(binary_tree)
Type: function
Returns the amount of nodes in <binary_tree>. See the chapter on binary trees for more information. Example:

```
PRINT "Amount of elements: ", TOTAL(mytree)
```

TRACE

TRACE <**ON**|**MONITOR** <var1, var2, ...>|**OFF**>
Type: statement
The ON keyword starts trace mode. The program will wait for a key to continue. After each key press, the next line of source code is displayed on the screen, and then executed. A double quote symbol will be replaced for a single quote, a back slash symbol will be replaced by a forward slash and a percentage symbol will be replaced by a hash symbol to avoid clashes with the C printf function. Pressing the ESCAPE key will exit the program.
The MONITOR keyword also starts trace mode, but allows monitoring values of variables. After each line of source code the content of the specified variables is displayed.
If TRACE is used within a function, make sure to also add TRACE OFF at the end of the function.
Example:

```
LOCAL var
TRACE MONITOR var
FOR var = 1 TO 10
    INCR var
NEXT
```

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.

TREE

TREE <binary tree> **ADD** <value> [**TYPE** <type>]
Type: statement
The TREE statement adds a value to a binary tree. Adding a duplicate value to the tree will not have any effect, and will silently be ignored. For more information and examples, see the chapter on binary trees. See also FIND to verify the presence of a node in a binary tree or TOTAL to determine the total amount of nodes in a tree. Example:

```
DECLARE mytree TREE STRING
IF NOT(FIND(mytree, "hello world")) THEN TREE mytree ADD "hello
world"
```

TRUE

TRUE
Type: variable
Represents and returns the value of '1'. This is the opposite of the FALSE variable.

TYPE

TYPE [**SET**|**UNSET**|**RESET**] [**BOLD**|**ITALIC**|**UNDERLINE**|**INVERSE**|**BLINK**|**STRIKE**]
Type: statement
The TYPE statement sets the font type in an ANSI compliant console. It is allowed to specify multiple types on the same line. Changes can be undone per type using the UNSET keyword. The RESET keyword will restore the default font settings. Note that not all Linux shells implement every console font type. See also the COLOR statement to set the color. Examples:

```
TYPE SET ITALIC BLINK
PRINT "This is blinking italic!"
TYPE UNSET BLINK
PRINT "Italic left!"
TYPE RESET
```

TYPEOF$

TYPEOF$(x)
Type: function
Returns the type of a variable.

UBOUND

UBOUND(array)
Type: function
Returns the total elements available in a static, dynamic or associative array. In case of multi-dimensional static arrays, the total amount of elements in the array is returned. Example:

```
LOCAL array[] = { 2, 4, 6, 8, 10 }
PRINT UBOUND(array)
```

UCASE$

**UCASE$**(x$)
Type: function
Converts x$ to uppercase characters and returns the result. See LCASE$ to do the opposite.

UCS

UCS(char)
Type: function

Calculates the Unicode value of the given UTF8 character (opposite of UTF8$). See also ASC for plain ASCII characters. Example:

```
PRINT UCS("©")
```

ULEN

ULEN(x$ [, y])
Type: function
Returns the length of the UTF8 string x$. Optionally, the position at y can be specified. See LEN for plain ASCII strings.

UNESCAPE$

**UNESCAPE$**(string$)
Type: function
Parses the text in <string$> and converts escape sequences into actual special characters (like newline or Unicode). This functionality comes handy when reading C text or JSON text containing escape sequences. Escape sequences for actual binary data (like '\x') are not converted. See also ESCAPE$ to do the opposite. Example:

```
OPTION UTF8 TRUE
PRINT UNESCAPE$("Hello\\nworld\\t\\U0001F600")
```

UNFLATTEN$

**UNFLATTEN$**(txt$ [, groupingchar$])
Type: function
Unflattens a string where the double quote symbol is used to group parts of the string together. The string will be surrounded with double quotes and any existing escapes will be escaped. Instead of the double quote symbol a different character can be specified (optional). See also FLATTEN$ for the reverse operation. Examples:

```
PRINT UNFLATTEN$("\"Hello \\\"cruel\\\" world\"")
PRINT UNFLATTEN$("\'Hello world\'", "'")
```

UNIQ$

**UNIQ$**(string$ [,delimiter$])
Type: function
Unifies all elements in a delimited string split by delimiter$. The delimiter$ is optional. If it is omitted, then the definition from OPTION DELIM is assumed. When specified, it may consist of multiple characters.
If delimiter$ occurs between double quotes in string$, then it is ignored. This behavior can be changed by setting OPTION QUOTED to FALSE. See also the chapter on delimited string functions for more information about delimited strings. Example:

```
PRINT "A sequence with unique members: ", UNIQ$("a a b c c d e f a
c f")
```

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
```

UTF8$

UTF8$(x)
Type: function
Returns the character belonging to Unicode number x. This function does the opposite of UCS. The value for x can lie between 0 and 0x10FFFF. Note that the result only will be visible in a valid UTF8 environment, and also that the installed character set should support the character. See also CHR$ for plain ASCII characters. The following should print a smiley emoticon:

```
LET a$ = UTF8$(0x1F600)
PRINT a$
```

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$>|<array#>|<array%>|<array> [**TYPE** c-type] **SIZE** <total>
Type: statement
Declares a variable argument list in a FUNCTION or SUB. There may be no other variable declarations in the function header. The arguments to the function are put into an <array> which is visible within the FUNCTION or SUB. The type can be defined either by the optional TYPE keyword or by a variable suffix. If no type is specified then VAR will assume NUMBER. The SIZE keyword defines where the resulting amount of elements will be stored. 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")
```

VERIFY

VERIFY(connection, file$)
Type: function
Verifies the certificates of the current TLS connection against file$. The file$ should be in PEM format and should contain all root CA certificates. Usually this information can be extracted from a web browser. See also the chapter on secure network connections.
In case of an invalid certificate, OpenSSL and GnuTLS will not drop the active TLS connection, and the VERIFY function will return the TLS error code.
For WolfSSL, the connection will be dropped immediately, and the reserved ERROR variable will contain the actual TLS error code. Use CATCH GOTO to prevent the program from stopping and workaround the TLS problem.

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 file descriptor (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.

WALK$

**WALK$**(directory$, filetype, regex, recursive [,delimiter$])
Type: function
This function returns a delimited string with all the file names located in <directory$>. The <filetype> argument can contain a number and determines the kind of file to look for. These values can be combined in a binary OR:

Value	Meaning
1	Regular file
2	Directory
4	Character device
8	Block device
16	Named pipe (FIFO)
32	Symbolic link
64	Socket

The <regex> argument defines a regular expression and acts as an additional filter to narrow down the resulting list further. The <recursive> argument can either be TRUE or FALSE to define whether or not underlying directories should be searched as well.
The delimiter$ argument is optional. If it is omitted, then the definition from OPTION DELIM is assumed. When specified, it may consist of multiple characters. Note that the default delimiter used by BaCon is a single whitespace, while file names can contain whitespaces as well. It is therefore recommended to specify a delimiter like NL$ which usually does not occur in file names.

If delimiter$ occurs between double quotes in string$, then it is ignored. This behavior can be changed by setting OPTION QUOTED to FALSE. See also the chapter on delimited string functions for more information about delimited strings.
Example to list files and directories in the current directory using a binary OR:

```
PRINT WALK$(".", 1|2, ".+", FALSE, NL$)
```

Example to list all directories in /tmp containing an underscore symbol:

```
PRINT WALK$("/tmp", 2, "[_]+", FALSE, NL$)
```

Example to recursively list all files ending in ".jpg" and also start with a number:

```
PRINT WALK$(".", 1, "^[[:digit:]]+.*.jpg$", TRUE, NL$)
```

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.

WHERE

WHERE(string$, position [,delimiter$])
Type: function
Returns the actual numerical character position in a delimited string split by delimiter$. The delimiter$ is optional. If it is omitted, then the definition from OPTION DELIM is assumed. When specified, it may consist of multiple characters. If delimiter$ occurs between double quotes in string$ then it is ignored. Example:

```
PRINT WHERE("a b c d \"e f\" g h i j", 6)
```

WHILE

WHILE <equation> [**DO**]
  <body>
  [**BREAK**]|[**CONTINUE**]
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. With the optional CONTINUE part of the body can be skipped. Example:

```
LET a = 5
```

```
WHILE a > 0 DO
    PRINT a
    a = a - 1
WEND
```
As the WHILE statement uses an equation to evaluate, it also allows the BETWEEN operator:
```
a = 2
WHILE a BETWEEN 1;10
    PRINT a
    INCR a
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. A semicolon at the end of the line prevents writing a newline. Refer to the PUTBYTE statement to write binary data. Examples:
```
WRITELN "Hello world with a newline" TO myfile
WRITELN "Without newline"; 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 value
6	Unable to claim memory
7	Unable to delete file
8	Could not open directory
9	Unable to rename file
10	NETWORK argument should contain colon with port number
11	Could not resolve hostname
12	Socket error
13	Unable to open address
14	Error reading from socket
15	Error sending to socket
16	Error checking socket
17	Unable to bind the specified socket address
18	Unable to listen to socket address
19	Cannot accept incoming connection
20	Unable to remove directory
21	Unable to create directory
22	Unable to change to directory
23	GETENVIRON argument does not exist as environment variable
24	Unable to stat file
25	Search contains illegal string
26	Cannot return OS name
27	Illegal regex expression
28	Unable to create bidirectional pipes
29	Unable to fork process
30	Cannot read from pipe
31	Gosub nesting too deep

32	Could not open device
33	Error configuring serial port
34	Error accessing device
35	Error in INPUT
36	Illegal value in SORT dimension
37	Illegal option for SEARCH
38	Invalid UTF8 string
39	Illegal EVAL expression
40	SSL file descriptor error
41	Error loading certificate
42	Widget not found
43	Unsupported array type

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

Appendix C: reserved keywords and functions

All keywords belonging to the C language cannot be redefined in a BaCon program:
auto, break, case, char, const, continue, default, do, double, else, enum, extern, float, for, goto, if, int, long, register, return, short, signed, sizeof, static, struct, switch, typedef, union, unsigned, void, volatile, while.
Functions defined in libc, libm or libdl cannot be redefined in a BaCon program, most notorious being:
exit, index, y0, y0f, y0l, y1, y1f, y1l, yn, ynf, ynl, dlopen, dlsym, dlclose.
Internal symbols and macro definitions cannot be reused. These start with '__b2c_'.
All symbols mentioned in the paragraph "Reserved Names" of any C manual cannot be redefined.

Appendix D: details on string optimization in BaCon

As BaCon is a plain Basic-to-C converter, the resulting code depends on the C implementation of strings, which, in fact, comes down to the known concept of character arrays. A string is nothing more than a series of values in memory which ends with a '0' value.
Plain C code is notorious for its bad performance on strings, especially when it needs to calculate the length of a string. Usually, the bytes in memory are verified until the terminating '0' value is encountered. For large strings this will take a considerable amount of time, especially in repetitive operations.
In the ongoing attempt to improve the performance of string operations, several approaches for the BaCon project have been investigated. The below report is a short summary of techniques which were tried.

Different binary layout

A common implementation stores the actual string length in the beginning of the string. This means that, for example, the first 4 bytes contain the length of the string, and the next 4 bytes contain the size of the buffer holding the string. (This would limit the size of the string to 4294967295 bytes (4Gb), which, for most purposes, should be enough.) After that, the actual characters of the string itself are stored.

b1	b2	b3	b4	l1	l2	l3	l4	char1	char2	char<n>
String length				Buffer length				Actual bytes of the string		

Programming languages like Pascal and Basic usually implement their strings this way. They change the binary layout of the actual string, where, as mentioned, the first 8 bytes contain meta information about length and buffer size.
For BaCon this seems a promising approach, however, as BaCon is a Basic-to-C converter, it makes use of the string functions provided by *libc*. I am referring to functions like printf, strcat, strcpy and friends. If strings are being fed to standard C functions like these, then this would cause unpredictable results. The libc string functions always assume character arrays in a correct format, otherwise they will not work properly - encountering binary values would cause printf to print garbage on the screen, for example.
Now, it is possible to add a default offset of 8 bytes to the beginning of the memory address to overcome this problem. However, we do not know in advance what type of string comes in. For example, the used string very well may contain a string *literal*. This is plain text hard coded in the BaCon program.

```
PRINT "Hello" & world$
```

In this example, both a string literal and also a memory address are used. Clearly, such string literal does not have the 8 bytes offset with meta information. So how can we check whether a string consists of a string literal, or whether it contains a memory address with the 8 byte offset?
I will come back to the concept of using a pre-buffer shortly.

Hash table

Another idea is to use hash values paired with the string length. The approach is to take the actual memory address of a string, which is a unique value. This unique memory address then can be put into a hash table which also can store a string length value (key-value store).
For performance reasons it would be nice if the table would provide values in a sequence starting from 1 to have indexes available for an array. This is called a “minimal” hash.

Furthermore, as we do not know in advance how many memory addresses the BaCon program is going to use, the hash table should be able to grow and shrink dynamically.
When it comes to performance, it turns out that has tables are too slow. The implementations of a dynamic hash table often make use of linked lists, and the unavoidable memory inserts and deletions simply take too much time. Eventually, it was established that they take more time than a string length calculation.

Pointer swap

In BaCon, all string operating functions make use of a temporary buffer to store their (intermediate) results. This is to ensure that nested string functions can pass their results to a higher level.

```
text$ = MID$("Goodbye, Hello", 9) & CHR$(32) & RIGHT$("The World", 5)
```

The above example concatenates three string operations into one resulting string. The functions MID$, CHR$ and RIGHT$ first will store their result into temporary buffers, which then are passed to the '&' operator, which, in its turn, stores the result of the concatenation into another temporary buffer. The final result then is assigned to the variable 'text$'. The assignment is performed by copying the contents from the temporary buffer to the variable 'text$'.
So, instead of *copying* the result into a variable, it also is possible to *swap* the memory addresses of the variable and the temporary buffer. The variable 'text$' will point to the result, and the contents of the temporary buffer will be changed to the previous character array of the variable. And because the buffer is temporary, it is going to be overwritten anyway in a next string operation, so there's no need to care about that.
This technique of pointer swapping will save some time, because it avoids a needless copy of bytes from one memory area to another. Even though it does not help us with string length calculation, it can help to improve the performance of string operations.

Static character pointers

When using character pointers in functions, it is a good idea to declare them as *static*. Such pointers keep their value, e.g. the memory address they are pointing to, so the next time when the function is called, the pointer still points to the allocated memory. Therefore, there is no need to free a pointer at the end of a function, or to allocate new memory when entering the function. This will save a lot of system calls and kernel activity.
Note that it may be needed to set the memory of the string to an empty string at the beginning of the function.
Also note that because static pointers always will point to the same memory, recursive functions may not work properly. So in each level of recursion, the same memory is overwritten.
The technique of static pointers is partly implemented in BaCon: static pointers are used for string variables and string arrays which get their value assigned at declaration time. This technique is not suitable for regular string variables, because of the aforementioned problems with recursion.

Using a pre-buffer

The last technique to discuss, which actually has been implemented, is using a pre-buffer. It is similar to the binary layout discussed earlier. In this approach, a block of memory is allocated, of which the first bytes contain meta information about length and buffer size. However, the returned pointer itself is a pointer to the start of the actual string. The benefit of this is that the string functions from libc will work properly, because they see the actual string.
As mentioned, this concept is used in BaCon. However, there is a severe problem which needs to be solved: how do we know if an incoming string already uses a pre-buffer?

```
PRINT a$ & b$ & c$ & "Hello"
```

In this example, we have three variables and a string literal. If one of the variables does not have a pre-buffer, and BaCon likes to look into it, then a crash will occur (segfault). BaCon cannot simply look into memory which is not allocated. Nor into a non-existing pre-buffer in case of a string literal.
To solve this problem, we have to think of a trick. The memory addresses which point to the strings are in fact numbers, each of which must be unique. If the addresses were not unique, then the program would overwrite one string with another. Therefore, the uniqueness of memory addresses is a property which can be used to identify a string with a pre-buffer: *if the address is taken as a plain number it can be used as an index in a lookup table*. That table contains a list of addresses for which BaCon has created a pre-buffer.
Obviously, such table cannot be as large as the amount of possible memory addresses. But this is not necessary. The biggest BaCon program at the time of writing is the implementation of BaCon in BaCon itself. It contains more than 10.000 lines of code, and when compiled with the '-@' debug parameter it can be observed that there are less than 400 strings in use, simultaneously, at any moment in time. It is therefore not realistic to implement a lookup table for all possible memory addresses, since it is unlikely that actual programs will use that amount of string data anyway. Currently, the size of the lookup table is 1Mb, which means that theoretically a program can identify a little over 1 million memory addresses during runtime.
The index in the table is determined by taking the modulo from the address by 1 million and then performing a binary 'AND' with '1 million minus 1'. This is a very fast algorithm but allows collisions to happen: some memory addresses will deliver the same index in the lookup table as other address will. The approach to solve this situation is called *linear probing*: in case of a collision, BaCon will take the next slot in the lookup table. If this slot is occupied with some other address, again it will take the next. The maximum probe depth currently is set to 16 steps. If the probe depth is exceeded then the program simply will generate an internal error and stop.
String operations now can lookup the address in the lookup table and quickly fetch information from each pre-buffer, saving time which otherwise would have been needed for length calculations.

<n> bytes	1	2	3	4	1	2	3	4	1	2	3	4	<n> bytes
Left buffer	String length				Size right buffer				Size left buffer				Right buffer
	<------- area containing metadata ------->												Actual string data
													---> Returned pointer

The above picture represents the new layout. Meta information is stored in an area which floats freely between two memory buffers. The returned pointer is the start of the right buffer which contains the actual string data. The left buffer is used for string concatenations, to insert text before an existing string, which allows a very fast concatenation of data (this works by first moving the area with meta data to the left and then inserting the text).
The parameters for the lookup table can be altered by the command line parameter '-t' (see the chapter on Basic Usage and Parameters).

Memory pool

As strings are stored in memory, it sometimes can be a good idea to allocate a block of memory before the actual program starts. A private function to share and administer can assign parts of that allocated memory to the program. This could save time, because the real memory allocation on the heap already took place, and the private memory allocation simply has to administer small chunks. Note that it only saves time in case a lot of subsequent allocations and frees take place in the program. Previously, BaCon performed almost 500,000 allocations consuming a total of 80 Mb of memory. A memory pool helps to decrease the allocations requested by BaCon to the kernel and it

also lowers the overall memory usage by reusing existing slots.
The core string allocation engine now uses a basic memory pool, which, by default, is defined by 2048 slots each occupying 1024 bytes. This decreases the amount of memory allocations with more than 85% and decreases the total memory usage with approximately 45%.
The parameters for the memory pool dimensions can be altered by the command line parameter '-t' (see the chapter on Basic Usage and Parameters).

Please report errors to: `REVERSE$("gro.retrevnoc-cisab@retep")`

Created with LibreOffice 7.3.7.2

Back to top of document