halec
Herrnröther Str. 54
63303 Dreieich
Germany

www.halec.de

halec
embedded solutions

Manual

roloBasic

Document version 1.0.368 as of 2012-03-09

Table of contents

I Getting started with roloBasic

Chapter I is a brief introduction of the features of roloBasic.

Chapter II explains the language features in detail.

Chapter III is the reference of the language syntax and the system functions.

1 What is roloBasic?

roloBasic is a simple programming language. It is executable on many microcontrollers in a self-sufficient way. A PC is not required for programming a microcontroller with roloBasic. A command line interface can be used for directly controlling a microcontroller. roloBasic source code can be processed and executed directly on a microcontroller.

roloBasic is implemented as a bytecode compiler and a virtual machine. This architecture allows very different configurations. If self-sufficiency is not important, the compiler can be omitted in the microcontroller. In this scenario the compiler can be used separately on a PC, while on the microcontroller only the VM is required.

roloBasic is frugal. About 32 kB flash and 1 kB RAM are enough for running the compiler and the VM. If only the VM is required, the requirements can be cut down even further.

roloBasic is extremely configurable. Different products can be supported by roloBasic in a very specific way. For example, roloBasic can be used only for flexible configurability of a product. On the other hand, a microcontroller application can be developed entirely in roloBasic. Very different scenarios are possible with roloBasic.

roloBasic can be configured as command line interface. This is called the roloBasic shell. It can be used to enter connected commands, which will be translated and executed directly thereafter.

2 Distinguished language properties

roloBasic is easy to learn. There are no over-complicated language structures, but still roloBasic is fairly expressive.

```
! Hello, world! in roloBasic:
print "Hello, world!"
```

2.1 Memory management

roloBasic uses automatic memory management for convenient programming. Arrays and strings can be created, removed and resized anytime, as long enough free memory is available. Memory is released by clearing references, for example by assigning the value 0 to a variable which holds an array:

```
a = dim(int, 7)  ! Int-Array with 7 elements. Index: 0-6
resize a, 12     ! Increase array size to 12 elements
print size(a)    ! print array size
a = 0            ! delete array and release memory
```

2.2 Exception handling

Exception handling allows convenient programming without waiving error handling.

```
print a[index]     ! print an array element
index = index + 1
print a[index]     ! print another array element

catch ex         ! catch possible exception and store in ex
                 ! if ex=0, no exception has been thrown
if ex=rangeCheckError
  print "Bad Index: ", index, " Array size is: ", size(a)
endif
```

II Language description

Before starting serious roloBasic programming, this chapter should be read entirely.

1 General language properties

roloBasic is a line oriented language. Most commands are terminated by an end of line. Within control structures line breaks are mandatory.

Lines can be connected using the underscore character ("_"). The compiler will treat lines connected in such a way like a single line.

```
print "This is a quite long line, which is " + _
  "partitioned into two lines."
```

The colon (":") can be used for placing more than one command in a single line. The compiler will treat every colon as an end of line:

```
if isArray(a) : print size(a) : endif
```

The exclamation mark designates the rest of a line as comment. Within a comment colons are ignored.

```
print "Hallo!"  !Comment: telling the obvious is bad style,
                !so we don't do it.
```

roloBasic is case insensitive: The capitalization of letters within keywords or identifiers does not matter. We use camelCase in source code examples for better readability, though.

2 Single line commands

A roloBasic program is defined by a sequence of commands. The most simple commands are procedure calls and assignments.

2.1 System functions

System functions are used to control input, output, and special properties or capabilities of a microcontroller driven device. Most system functions are device dependent and only available in specific devices.

However, the most important system functions are always available. This set of functions is called the roloBasic standard environment, which is documented in chapter III. Many functions of the standard environment are also explained within their scope of application.

Product specific system functions are not covered by this document. Please read the product instruction manual to learn about these functions.

Example for a system function of the standard environment:

```
print <value> [, <value 2> [, ...]]

Example:
=========
print "The answer of the ultimate question: ", 42
```

This system function outputs one or multiple values using the standard output. The standard output could be a display, a log file, a serial interface, or something completely different.

2.2 Variables and assignments

The most important roloBasic command is the assignment, which is denoted by the equality sign ("="):

```
<target> = <expression>
```

The target is usually a variable. Any value can be assigned to any variable, since roloBasic is using a dynamic type system.

It is important to understand that an assignment just assigns a new "name" to a value. The value itself will *not* be copied.

Variables don't have to be declared. The roloBasic compiler will create new variables automatically depending on usage. New variables will always be initialized with the value 0:

```
print newName ! Here the new Variable "newName" is created
              ! and printed. This results in printing the
              ! value 0.
```

2.3 Expressions and operators

Within a roloBasic assignment, everything which may be placed on the right side of the equality sign is an expression. roloBasic knows the following types of expressions:

```
a = "Hallo"         ! literal
b = a               ! variable
a = b + " Welt"     ! operator usage
b = size(a)         ! function call
b = (b+1)*2         ! bracket term
```

The most important operators are part of the standard environment.

3 Types and values

Because types are managed automatically, there is usually no need to care about them. But even though it is usually not noticeable, every value is always of a well defined type. In roloBasic types are assigned to values, not to variables.

3.1 Primitive data types

roloBasic knows just three primitive types:

- char: value range 0 ... 255
- int: value range -32768 ... 32767
- long: value range -2147483648 ... 2147483647

Dynamic typing

An unambiguous type is automatically assigned to every value, for instance:

```
type(7)  = char
type(-7)  = int
type(40 * 1000)  = long
```

The type assignment is automatic and mostly invisible. For every value always the same type will be assigned. So the expression `1001 - 1000` yields the result `1`, which is of type `char`.

Literals

In roloBasic all numeric literals are nonnegative. The "-" Operator can be used to represent negative numbers.

The decimal numbers 0 to 2147483647 may be used as numeric literals. Hex values are indicated by the "$" character.

```
a = -2147483647 - 1   ! assign the value -2147483648 to a
b = $1ff              ! assign the value 511 to b
```

`The type char` is used for both characters and positive numbers in the range of 0 to 255. Character literals are surrounded by single quotes:

```
'A' = 65
```

For character literals, the following escape sequences are available:

```
'\\'    : backslash (\)
'\''    : single quote
'\"'    : double quote
'\n'    : LF (line feed)
'\r'    : CR (carriage return)
'\t'    : tab character
'\b'    : backspace
'\f'    : form feed
'\0'    : 0 (zero value)
'\x??' : hex code (? represents a hex digit)
'\d???': decimal code
```

Unsupported basic types

In order to keep roloBasic small and economical, the type system was kept quite simple. Only a few basic types are available.

The standard environment does not support float values, since many microcontrollers don't offer native support for it. A software float library would be too resource demanding for small microcontrollers.

There is no boolean type as well. Truth values are represented by numbers: roloBasic interprets 0 as "false", and all other values as "true". All operators which compute truth values return either 0 or 1.

There is no type for representing types. The system function `type(<value>)` returns a number encoding a type. These values are given by the system constants `char`, `int`, `long` and `vari`.

3.2 Arrays

The memory management of roloBasic offers convenient array handling. An array is just an ordinary value like a number, and can be assigned to any variable or passed to functions. However, unlike numbers, arrays are mutable. The system function `dim` creates and returns arrays.

Mutability

Probably it is not apparent that a numeric value is immutable. However, assigning a value to a variable does not mean that the old value will be modified – it will rather be replaced as a whole. On the other hand, arrays can be modified without being replaced.

```
a = dim(int, 10)     ! Create array for 10 int values
a[0] = 30000         ! modification 1: write value into array
resize a, 20         ! modification 2: expand array size to 20

! Now the following equations are considered true:

! size(a) = 20
! isArray(a) = 1     ! (1 means "true")
! type(a) = int
! mutable(a) = 1
```

There are other ways to create arrays as using the system function `dim`. Several other system functions also can return arrays. Often these arrays will be tagged as immutable. The mutability of a value can be inquired with the system function `mutable`.

It is impossible to modify immutable values. The advantage of immutables is the ability of conflict free re-use for different purposes.

Element types

In roloBasic, arrays have a fixed element type, which cannot be modified after creation. The element type of an array determines which values the array can hold – and how much memory the array requires per element.

Assigning an illegal value to an array element results in the exception `typeFault`. The element type of an array can be inquired using the system function `type(<array>)`. Whether a value is an array or not, can be inquired by `isArray(<value>)`.

Size and index range

The index range of an array always begins with 0. The last valid index of an array `a` is always: `size(a)-1`. Any attempt to access an invalid array index will throw the exception `illegalArrayIndex`.

When creating an array using `dim` the size has to be specified. However, the array size still can be modified later using the command `resize`. Empty Arrays with size 0 are legal. The maximum size of an array is only limited by the availability of free memory.

Assignment is not copy

Whenever an array is being assigned to a variable, it will not be copied. Instead, the variable will be set as a new name for the array. In order to create a true copy of an array, the system function `copy` should be used.

```
a = dim(int, 10) ! Create an int array of size 10
b = a            ! b and a are now names of the same array
b[0] = 77
print a[0]       ! output 77
b = copy(a)      ! assign a copy of a to b
b[0] = 42
print a[0]       ! still output 77
```

Literals and strings

In roloBasic there is no separate type for strings. Instead, char arrays are used for this purpose. So, a string literal is really a literal for immutable char arrays:

```
s = "Hello roloBasic!"

! Now the following equations are true:
!          isArray(s) = 1 !(true)
!          type(s) = char
!          mutable(s) = 0 !(false)
!          s[0] = "H"
```

In string literals the same escape sequences can be used like in char literals.

The "+" operator can be used as well with numbers as with arrays. When using it with arrays, it creates a new array which contains the concatenation of the two original arrays. So this is a very convenient way of string concatenation. However, it is not very efficient. It is better to use the system function `append` for this purpose. To append single values to an array, the system function `push` can be used.

Creating mutable copies

Hint: The system function `copy` always creates mutable arrays. Whenever a modified version of an immutable array is required, it can be copied with `copy` and modified accordingly.

roloBasic also offers array constructors. These can be invoked with an arbitrary number of arguments:

```
a = int(0,1,2,3) ! creates an immutable int array with size 4
                 ! containing the values 0,1,2,3
b = copy(long()) ! creates an empty mutable long array
```

Hint: The compiler tries to make computations in advance (at compile time), if possible. For example, it pre-computes array creation with array constructors, and array concatenation. All pre-computed arrays are immutable. Pre-computation of system functions enabled for it will succeed, if all parameters are immutable and available at compile time.

The advantage of pre-computation and immutable data is memory economy. The disadvantage is, that some care has to be taken for it while programming. The most efficient way for creating a pre-initialized mutable array is using the `copy` system function. A less efficient way: Whenever a variable occurs within an expression, it will generally be computated at runtime only, and the result will be mutable.

```
a = copy(char(0,1,2,3,4,5,6,7)) ! Efficient way for
                                ! initializing a mutable
                                ! char array

c = 0                           ! Inefficient way for doing so
b = char(c,1,2,3,4,5,6,7)       ! ...
```

3.3 Vari arrays, structures

In roloBasic there exists another specialized array type, which can hold any kind of values – even other arrays.

This array type is called `vari`.

In roloBasic, vari arrays can be used as structures. Always when structures are mentioned, we are really talking about vari arrays. Using vari arrays, complex nested or even cyclic data structures can be created easily:

```
a = dim(vari,0)
push a, 4215                  ! fill the array a value by value
push a, "Königswasser"
push a, int(1,2,3)
```

Vari arrays can be created using array constructors, too:

```
a = vari("A", "few", "strings", dim(char,2))

! Now the following equations are true:
!           a[2] = "strings"
!           a[2][0] = 'S'
!           mutable(a) = 1 ! (mutable because of dim call)
```

A vari array slot works like a variable. An assignment to an array slot is not a copy. This is why chained or cyclic data structures can easily be created:

```
a = copy(vari(1,0))
b = copy(vari(2,a))
a[1] = b

! now the following equations are true:
!                a[1][1][0] = 1
!                a[1][1][1][0] = 2
```

3.4 Equivalence and identity

The system function `same` checks two values for identity. This is a very strong criteria. Two numbers with the same value are always identical, but an array is only identical to itself. Two individually created arrays are never identical, even if they contain the same values. An exception is when two immutable arrays with same values are created. In this case, it can happen that the array is only created once – so it will be identical to itself, of course.

The relational operator "=" checks two values for equivalence. This is a weaker criteria. Two equally sized arrays are equivalent, if they share identical values in the same order. This is even true if the element type of the arrays is different. For example, an int array can be equivalent to a long array:

```
! Same numeric values are always identical:
! e.g. same(10,10) = 1

a = vari(1,2)
b = int(1,2)
c = a

! now the following equations are true:
!   same(a,b) = 0   ! a and b are not identical...
!   a = b           ! ...but equivalent
!   same(a,c) = 1   ! a and c are Names for the same
!                   ! array, so same(a,c) = 1

a1 = vari(a,3)   ! Now we create two nested structures holding
b1 = vari(b,3)   ! the same values

! now the following equations are true:
!   a1 <> b1  ! a1 and b1 are neither equivalent nor identical
```

This may be surprising at first glance. However, the elements of `a1` and `b1` are not identical, so `a1` and `b1` are not equivalent: `a1` and `b1` can be distinguished just by comparing their elements using `same`:

```
! clearly the following equations are true:
!   same(a1[0],a) = 1
!   same(b1[0],a) = 0
! thus a1 and b1 are not equivalent.
```

4 Control flow statements

roloBasic has only few control flow statements. The GOTO statement, well-known from other basic dialects, is not implemented in roloBasic.

4.1 Conditional statements

roloBasic uses the IF-Statement for branching. The associated keywords are `if`, `endif`, `else` and `elseif`. Directly after `if` and `elseif` an expression followed by a line break is expected. After `else` and `endif` a line break is expected, too:

```
if isArray(x) or isArray(y)
  print "x or y is an array."
elseif x>y
  print x, " is greater than ", y
elseif x=y
  print x, " equals ", y
else
  print x, " is smaller than ", y
```

```
endif
```

4.2 Iteration

Iteration is usually much more efficient as recursion.

For loop

The for loop is convenient, but not the most efficient control statement for iteration:

```
a = "Hello roloBasic!"
b = dim(char,0)

for i=0 to 4
  push b, a[i]+1
next

for i=size(a)-1 downto 0 step -2
  push b, a[i]
next

! Now we have: b = "Ifmmp!iaoo le"
```

Watch out: The step and stop values are recalculated every iteration.

The keyword `downto` can be used to decrement the loop counter. The keyword `step` is required for increment values different from 1 or -1.

Do loop

Die infinite loop looks like this:

```
do
  println "... till hell freezes over..."
loop
```

The `do` can be followed by the keyword `while` and an entry condition. The `loop` can be followed by the keyword `until` and an exit condition. Any loop can be aborted by the `break` command.

```
! Loop with 3 exit paths:
do while y < x*x
  y = getnextvalue(y)
  if y = check
    break
  endif
loop until y < max
```

5 Procedures and functions

Frequently used command sequences or calculations should be implemented as a procedure or function.

5.1 Procedures

A procedure definition is initiated with the keyword `procedure`, followed by the procedure name and a comma separated list of formal parameters. A line break at the end of this list finishes the procedure head.

The subsequent commands define the body of the procedure. The procedure is finished by the keyword `end`.

```
procedure printarray a
  if isarray(a)
    print "("
    do
      printarray a[i] : i=i+1
      if i=size(a) : break : endif
      print ","
    loop
    print ")"
  else
    print a
  endif
end

printarray "Hello" ! --> (72,101,108,108,112)
```

In roloBasic, Parameters are always value parameters. They work like local variables, which are initialized by the actual parameters every time the procedure is called.

Local variables are always initialized by 0 every time the corresponding procedure is called. Within a procedure, all variable references are considered local by default. So in the example code, the variable `i` is created automatically as a local variable and initialized with the value 0 every time the procedure `printarray` is called.

Within a procedure, global variables can be accessed using the prefix keyword `global`:

```
procedure log message
  append global logstring, message
end

log "Hello"
log ", "
log "roloBasic"
log "!"

! --> logstring = "Hello, roloBasic!"
```

Procedures are ordinary values, like arrays. A procedure definition works by creating a vari array which contains a representation of the procedure, and assigning it to a variable, whose name is given by the procedure name. So, procedures may be copied, stored in vari arrays or used as actual parameters for other procedures.

Watch out for the following characteristics:

- In order to call a procedure, it has to be stored within a variable. The roloBasic syntax doesn't allow calling a procedure which is stored as array element directly. However, the procedure may be copied into a variable in order to be called.
- Within a procedure, variable references are local by default. In contrast, procedure calls are global by default. So, a procedure call within a procedure will access a global variable, which contains the procedure to be called. In order to call a procedure which is stored within a local variable, the prefix keyword `local` has to be used.
- At least one space character is required between a procedure name and its first parameter.

```
procedure test1
  print "hello"
end

procedure test2 p
  proc = p[0]       ! copy procedure from p[0] into local var.
  local proc        ! call the procedure
end

a = dim(vari, 1) ! create vari array with a single element
a[0] = test1     ! copy procedure test1 into array a
test2 a          ! call procedure test with actual parameter a
```

When execution arrives the end of a procedure body, the procedure is terminated. It is also possible to terminate a procedure at any point within its body using the command `return`.

A procedure has no direct return value. However, values may be returned by using global variables. It is also possible to pass an array as actual parameter, and modify it within the procedure. So, an array may be used as a way for returning several values from a procedure call.

5.2 Functions

A function works almost like a procedure. However, a function always returns a value. Furthermore, the parameter list of a function is always delimited by parentheses.

The return value may be passed using the command `return <expression>`. Within a function – in contrast to a procedure – the keyword `return` always has to be followed by the return value. If execution arrives the end of a function body without coming across a `return` command, the value 0 will be returned.

Functions are values, and exactly like procedures, they may be assigned to variables or arrays elements. However, unlike procedures, functions contained by array elements may be called directly.

```
function fib(x)
  if x<=0
    return 0
  elseif x=1
    return 1
  else
    return fib(x-2) + fib(x-1)
  endif
end

procedure test2 p
  print p[0](10) ! call the function contained in p[0]
end

print fib(10)     ! print 55
a = dim(vari, 1)  ! create vari array with a single element
a[0] = fib        ! copy function fib into array a
test2 a           ! call procedure test2 with actual parameter a
                  ! --> will print 55, too
```

6 Exceptions

In roloBasic, a runtime errors always throws an exception, which can be caught using the `catch` command. An uncaught exception results in program termination, furthermore an error message containing the exception value will be generated.

The `throw` command is used for throwing user exceptions. These work exactly like system exceptions. If the value of a system exception is passed to the throw command, there will be no difference between the user generated exception and an original system exception.

```
i = 5
a = 1000
do
  a = a div i
  i = i - 1
loop

catch ex
if ex = divisionByZero
  print a
else
  throw ex    ! If it was a different exception, we just
              ! throw it again.
endif
```

The `catch` command must be followed by a single variable name. It will catch any exception which is "passing by" and assign it to this variable. If a `catch` command is arrived without any exception, the value 0 will be assigned. So, exception handling will usually look like this:

```
catch ex
if ex
  ... error handling ...
endif
```

The `throw` command throws an exception. It accepts an arbitrary value as parameter, which will be assigned to the exception. Even nested data structures may be used as value of an exception. However, if an exception is not caught by a `catch` command, the system will only output single numeric values within the resulting error message.

III The standard environment

All operators, procedures, functions and constants specified within this section are integral parts of roloBasic. Their names are reserved, and cannot be used as variable names. It is not possible to assign new values to these names. System functions and procedures can be called exactly like user defined ones. However, they do not represent values, so they cannot be passed as parameters or assigned to variables or array elements.

1 Arithmetic operators

Small values need less memory as big values. This memory "micro management" for numeric values works fully automatic, so the user doesn't have to care about it. The user may assume having 32 bit values all the time, with values within the range -2147483648 ... 2147483647. However, numeric overflows beyond this range won't be detected. So the expression `2147483647 + 1` yields the result -2147483648.

1.1 Addition

```
value = a + b
```

Adds two values.

- **Parameters:** Two numeric values or two arrays with same element type.
  The following condition must hold:
  `(not isarray(a) and not isarray(b)) or _`
  `isarray(a) and isarray(b) and type(a)=type(b)`
  Violation of this condition triggers the exception **typeFault**.
- **Return value:** Result of the addition. If the parameters are arrays, a new array of the same element type will be created, which contains the concatenation of both parameter arrays.
  (`"hi"+"ho"="hiho"`)

1.2 Subtraction, Multiplication

```
value = a - b
value = a * b
```

Operators for substraction and multiplication.

- **Parameters:** Two numeric values. The following condition must hold:
  `not isarray(a) and not isarray(b)`
  Violation of this condition triggers the Exception **typeFault**.
- **Return Value:** Result of the computation.

1.3 Division, Modulo

```
value = a div b
value = a mod b
```

Calculates the integer division and the remainder accordingly.

- **Parameters:** Two numeric values. The following condition must hold:
  `not isarray(a) and not isarray(b)`
  Violation of this condition triggers the Exception **typeFault**. If the value of the second parameter is 0, the Exception **divisionByZero** will be thrown.
- **Rückgabewert:** Result of the computation.

2 Relational operators

A common characteristic of all relational operators in roloBasic is that they are not recursive. Nested structures will only be examined at top level. Array elements containing other arrays will only be compared in terms of identity.

All comparisons result in a truth value. roloBasic uses the values 0 (as "false") and 1 (as "true") to represent truth values. roloBasic interprets all values unequal 0 as "true".

2.1 Identity (same)

```
value = same(a,b)
```

Determines the Identity of two values, that is, if `a` and `b` are the same value. This is always true for two numeric values if they represent the same number.

However, an array is only identical to itself. Two seperately created arrays with the same content are *not* identical.

If `a` is an array and `same(a,b)` holds, changing an element of `a` will lead to the same change in `b`, since `a` and `b` are just different names of the same data object.

Assigning a new value to `a` will not lead to any change in `b` because in this case just the meaning of the name `a` is modified. The data object formerly assigned to `a` is *not* modified.

- **Parameters:** Two arbitrary values.
- **Return value:** 0 or 1

2.2 Equivalence (=, <>)

```
value = a = b
```

Determines the equivalence of two values.

```
value = a <> b
```

Determines if two values are *not* equivalent.

The following assertion always holds: `same(a=b, not (a <> b))`

In roloBasic, two arrays are equivalent, iff they are of same length and all contained values are pairwise identical.

The following function computes the equivalence of two values by using the system function `same`:

```
function equivalent(a,b)
  if not isarray(a)
    return same(a,b)
  elseif same(size(a), size(b))
    for i=0 to size(a)-1
      if not same(a[i], b[i])
        return 0
      endif
    next
    return 1
  else
    return 0
  endif
end
```

The assignment operator "=" uses the same symbol like the equivalence operator, however this doesn't matter much.

After an assignment

```
a = b
```

always holds: `same(a,b)`. So, an assignment does more than just establishing equality. In contrast, after this assignment, `a` and `b` are names of the same data object.

- **Parameters:** Two arbitrary values.
- **Result value:** 0 or 1

2.3 Order relations (>,<,>=,<=)

```
value = a > b
value = a < b
value = a >= b
value = a <= b
```

These operators all define the same order. Therefore always holds:

```
(a < b) = (b > a)
(a < b) = (not (a >= b))
(a < b) = (not (b <= a))
```

Strictly speaking, they define two orders:

- Numerical order: The order relations can be used to compare numeric values.
- Lexicographical order, relating to arrays containing numerical values.

When comparing arrays using order relations, the arrays must contain only numeric values. Failing this condition triggers the exception **typeFault**.

Anyhow, this means that strings may be compared using order relations, since they are represented by char arrays, and char is a numeric type in roloBasic. Therefore holds:

```
"dwarf" < "dwelling"
```

- **Parameters:** Two numeric values, or two arrays containing only numeric values.
- **Result Value:** 0 or 1

3 Logic and bit manipulation

All logic operators except `not` interpret numbers as bit fields. They can also be used for boolean terms, however, only bit 0 will be used in this case.

This is easier as it sounds:

- For writing boolean terms, the operators `and`, `or`, `xor` and `not` may be used.
- For bit field manipulation, the operators `and`, `or`, `xor`, `<<`, `>>` and `bnot` maybe used. All numeric values should be regarded as 32 bit values in two's complement representation.

3.1 and, or, xor

```
value = a and b
value = a or b
value = a xor b
```

bitwise logic operations in 32 bit two's complement. May also be used as boolean operators.

- **Parameters:** Two numeric Values. The computation always works as if all values were 32 bit, even if the input values are small numbers.
- **Return value:** Result of the bit manipulation.

3.2 Boolean negation (not)

value = not a

Boolean negation.

- **Parameters:** A numeric value.

- **Return value:** If `a=0`, the return value is 1, otherwise it is 0.

3.3 Negation for bit manipulation (bnot)

```
value = bnot(a, type)
```

Not operation for manipulation of bit fields. The value `a` is regarded as bit field of the size of the type `type`. This is 8 bits for the type `char`, 16 bits for `int` and 32 bits for `long`. All bits in this field will be negated. The result of the operation is a numerical value of the specified type.

- **Parameters:** A numeric value and a numerical type code.
- **Return value:** Result of the bit field negation.

3.4 Arithmetic shift operators (<<, >>)

```
value = a << b
```

Shifts all bits of `a` by `b` steps to the left. This is equivalent to a multiplication of `a` with 2^b.

```
value = a >> b
```

Shifts all bits of `a` by `b` steps to the right. The most significant bit (and hence the sign) is retained.

- **Parameters:** Two numeric values.
- **Return Value:** Result of the shift operation.

4 Array management

Arrays are the only kind of data structures in roloBasic, and they are very efficient. Arrays can be created at any time, as long enough free memory is available. The memory taken by an array can be deallocated by simply "forgetting" the array, by making sure no more access to the array is possible. Arrays can be resized anytime, the maximum array size is only bounded by the amount of free memory available. Arrays can also be empty, and therefore contain zero elements. The array content is retained when resizing, except elements removed due to downsizing. When upsizing, new elements are initialized by 0.

Arrays have a fixed element type. The possible element types are:

```
char, int, long, vari
```

The element type determines which values are allowed to be stored in an array. `char`, `int` and `long` permit numeric values within the corresponding range only. `vari` allows arbitrary values, even arrays. Therefore, vari arrays are often called structures.

These element type names are implemented as system constants encoding a numeric code which represents the corresponding type. However, they can also be used as array constructors, and be called like a function with variable argument list. The parameters of the constructor call determine the initial content of an array created this way. Array constructors create immutable arrays, if all values are known at compile time.

4.1 Creating mutable arrays (dim)

```
value = dim(type, size)
```

Creates a new, mutable array of type `type` and size `size`. All Array elements are initialized as 0.

- **Parameters:** A numeric type code and a numeric size value.
- **Return value:** The newly created array.
- **Exceptions:** outOfMemory.

4.2 Array constructors(char, int, long, vari)

```
value = char(...)
value = int(...)
value = long(...)
value = vari(...)
```

Creates a preinitialized immutable array with the respective element type.

- **Parameters:** Arbitrary list of values. The values must comply with the element type. The list also may be empty in order to create an empty immutable array.

- **Return value:** The newly created immutable array.
- **Exceptions:** outOfMemory.

4.3 Array identification(isArray)

```
value = isArray(a)
```

Determines whether `a` is an array or not.

- **Parameters:** An arbitrary value `a`.
- **Return value:** 1, if `a` is an array. Otherwise 0.

4.4 Array resizing (resize)

```
resize a, newSize
```

Alters the size (=Number of elements) of the array `a` to `newSize`. Newly created array elements are initialized by 0. Mutable arrays may be resized anytime. However, immutable arrays cannot be resized. The least possible array size is 0. The maximum size is only determined by the amount of free memory available.

- **Parameters:** An array and a numeric value.
- **Exceptions:** outOfMemory.

4.5 Inquiring array size (size)

```
value = size(a)
```

Determines the number of data elements of `a`. If `a` is an array, `size(a)` is the current number of data elements. If `a` is a numeric value, `size(a)` = 1.

- **Parameters:** An arbitrary value, usually an array.
- **Return value:** The number of data elements of a.

4.6 Creating empty mutable arrays (reserve)

```
value = reserve(type, size)
```

Creates a mutable empty array of type `type`, which is prepared for very efficient resizing up to `size` elements. The `reserve` command is especially useful for creating arrays which are going to be used as a stack. (Resizing beyond `size` is still possible. And arrays which are created with `dim` can be used as a stack too.)

- **Parameters:** A numeric type code and a numerci value.
- **Rückgabewert:** The newly created empty array.
- **Exceptions:** outOfMemory.

4.7 Using arrays as a stack(push, pop, top)

```
push a, x
value = pop(a)
value = top(a)
```

In roloBasic arrays can be used as a stack.

- The command `reserve` is useful for preparing an empty stack.
- The command `push` appends the value `x` as new last element to an array `a` and increases the array size by 1.
- The function `top` determines the last Element of an array `a`.
- The funktion `pop` reads the last Element and removes it from an array `a`.
- **Parameters:** `a` must be an array. `x` may be an arbitrary value which is admissible as array element of `a`.
- **Return value (only for pop and top):** The last element of the array `a`.
- **Exceptions:** outOfMemory.

5 More data management commands

Some additional commands support the usage of roloBasic data objects.

5.1 Copying of data objects (copy)

```
value = copy(a)
```

Creates a copy of `a`.

If `a` is an array, the copy will be mutable.

Only a shallow copy is created. The copy contains exactly the same elements as the original. The elements of an array will not be copied, but assigned.

- **Parameters:** `a` may be an arbitrary value, but basically only the use of array is reasonable.
- **Return value:** A copy of `a`.
- **Exceptions:** outOfMemory.

5.2 Inquiring the element type (type)

```
value = type(a)
```

Determines the element type of `a`. If `a` is a numeric value, its element type is only dependent on its value. Therefore holds:

```
! type(0)  = char
! type(256)  = int
! type(-1)  = int
! type(100000)  = long
```

If `a` is an array, the element type which was specified by its creation(e.g. with `dim`) will be returned.

- **Parameters:** An arbitrary value
- **Return value:** The numeric element type code representing the element type of `a`.

5.3 Inquiring mutability(mutable)

```
value = mutable(a)
```

Determines whether `a` is mutable or immutable. Immutable data objects cannot be modified. So, immutable arrays cannot be resized, and it is not possible to assign values to array elements of immutable arrays.

- **Parameter:** an arbitrary value

- **Return value:** 1 if `a` is mutable, else 0.

6 Constants

In roloBasic several global system constants are defined. The names of these constants are reserved and cannot be used as variable names. These system constants contain numeric values, for example for encoding types and exceptions.

6.1 Type codes

roloBasic defines codes for 4 types:

```
char
int
long
vari
```

These constants are codes for the basic types of roloBasic. They can be used for comparing types or specifying element types of arrays. The type `vari` has an exceptional position, as it can only be used as element type of arrays.

```
! Example:
if type(a) = char and isArray(a)
  print "a ist a string"
endif
```

6.2 Error codes

Whenever a runtime error occurs, an exception will be thrown, which is represented by a numeric value. These values are defined by the following system constants:

```
outOfMemory          ! Not enough free memory available

rootstackOverflow    ! Internal system error
nullpointerAccess    ! Internal system error

valueRange           ! Range overflow. For example by
                     ! assignment of too large values arrays of
                     ! element type char or int, or by calling
```

```
                   ! system functions with restricted value
                   ! range for their parameters.

divisionByZero     ! Division by 0. May happen with div, mod

argumentFault      ! Illegal number of parameters when calling
                   ! a system function

illegalFunction    ! A variable was used as name for a function
                   ! within a function call, although it does
                   ! not contain a valid function.
                   ! Also applies for procedure calls.

indexRange         ! Index range overflow during array access

typeFault          ! Type error.
                   ! e.g. when calling a system function which
                   ! excepts a numeric value as argument with
                   ! an array.

! Example:

print "Integer Quotient: ", a div b
print "Remainder: ", a mod b
catch x
if x = divisionByZero
  print "Division by zero!"
endif
```

IV Appendix

Tables and formal specifications for roloBasic.

1 Syntax (in EBNF)

```
Program   = { ( Statement | Procedure | Function ) NewLine } .
Procedure = "procedure" Ident FormalParams NewLine Block "end" .
Function  = "function" Ident "(" FormalParams ")" NewLine
            Block "end".

Block       = { Statement NewLine } .
Statement   = Assignment | ProcCall | Do | For | If .
Assignment  = Lvalue "=" Expression .

VarAccess   = [ "global" | "local" ] Ident .
Lvalue      = VarAccess | ArrayAccess .
ArrayAccess = Expression "[" Expression "]" .

Expression      = FunCall | UnaryOperation | BinaryOperation |
                  Literal | VarAccess | ( Expression ) .
UnaryOperation  = UnaryOperator Expression .
BinaryOperation = Expression BinaryOperator Expression .

ProcCall     = Ident ActualParams .
FunCall      = Ident "(" ActualParams ")" .
FormalParams = [ { Ident "," } Ident ] .
ActualParams = [ { Expression "," } Expression ] .

Do =  "do" [ "while" Expression ] NewLine
      Block
      "loop" [ "until" Expression ] .

For = "for" Ident "=" Expression "to" Expression
      [ "step" Expression ] NewLine
      Block
      "next" .

If =  "if" Expression NewLine
      Block
      { "elseif" Expression NewLine Block }
      [ "else" NewLine Block ]
      "endif" .
```

1.1 Terminals

- Ident: Identifier. Only the characters A-Z, 0-9, and underscore ("_") may be used. May not begin with a digit.
- NewLine: Line Break or colon (":").
- Literal: Representation for a value constant. Examples: `'x' "Hello" -333 0.127`
- UnaryOperator: Unary operator, e.g. negation.
- BinaryOperator: Binary operator, e.g. addition.

Within string literals the backslash character ("\") is used to represent special characters, e.g. : `\" \\ \n \t \064`

1.2 Operator priorities

Operator symbol	Priority	Remark
- , not	8	unary
* , div , mod	7	mod, div: yields integer value
+ , -	6	+ may also be used with strings, arrays
<< , >>	5	shift operations
< , <= , > , >=	4	also compares strings, arrays
= , <>	3	also compares strings, arrays
and	2	also works bitwise
xor, or	1	also works bitwise

2 Types, Memory usage of values

Type	Memory usage (bytes)	As array element (bytes)
char	2	1 (in char arrays)
int	4	2 (in int arrays)
long	6	4 (in long arrays)
array of char (string)	2+size(...)	4 + size(...)
array of int	2+size(...)*2	4 + size(...)*2
array of long	2+size(...)*4	4 + size(...)*4
array of vari (structure)	2+size(...)*2	4 + size(...)*2