BestBASIC

Introduction

v 1.6

by Mr. Kibernetik

Language Basics

BestBASIC is an extension of Minimal BASIC.

BestBASIC keeps an attractive simplicity of Minimal BASIC, significantly expands its function set and adds much more power and flexibility to the BASIC language syntax.

Variables

There are no restrictions which characters can be used in variables names except those characters which form language syntax: brackets, math operators, space, dot, comma, semicolon, quote and apostrophe.

There are different numeric formats which use some letters or symbols, so this should be considered when creating a variables.

There is a simple rule: if variable name cannot be misunderstood then it is fine. All these variables names are valid:

```
N, x1, 12val, текст, 颜色, W$$, $1, row#, a&b, §12
```

Latin alphabet letters are not case sensitive, but characters from other alphabets are case sensitive.

◇ ◇ ◇

A value can be stored in a variable using the assignment operator '=':

```
n = 1
```

It is possible to assign the same value to multiple variables in one operation. The following code sample:

```
a = 0
b = 0
c = 0
```

can be written as:

```
a b c = 0
```

It is possible to assign different values to multiple variables in one operation. The following code sample:

```
a = 10
b = 20
c = 30
```

can be written as:

```
a b c = 10 20 30
```

◇ ◇ ◇

Variables have no fixed data type. Any variable can store numeric or text value.
This program:

```
a = 5
PRINT a
a = "text"
PRINT a
```

outputs:

```
 5
text
```

Code writing

There is no much difference between comma '`,`' and space '` `' separators when writing program code. These separators are interchangeable and it is up to you which separator to use.

All these code lines are exactly the same:

```
a,b,c = 1,2,3
a,b,c = 1 2 3
a b c = 1 2 3
```

Although space can be used as a separator in many cases, there is one important note: space does not separate in front of a math operator and in front of a bracket.

This is an example of a missing argument because space does not separate in front of a math operator. This code:

```
DRAW LINE 0,0 -100,100
```

is processed as:

```
DRAW LINE 0,0-100,100
```

and finally there are only three arguments instead of four.

This is another example of a missing argument because space does not separate in front of a bracket. This code:

```
DRAW LINE x,y (x+n),(y+m)
```

is processed as:

```
DRAW LINE x,y(x+n),(y+m)
```

and finally there are only three arguments instead of four.

So, if you use space as a separator be sure that it is enough to separate. And you will need to use comma if arguments collapse.

◇ ◇ ◇

Function arguments can be specified without brackets if it does not cause misunderstanding.

All these code lines are correct:

```
PRINT 2+2
PRINT(2+2)
PRINT
PRINT()
```

There is an important note regarding syntax analysis of functions which are called individually and functions which are used as parts of expressions.

If a function is called individually then it always takes the whole code line. This means that everything after the function name are the arguments of this function:

```
PRINT (a+b)*2, c+d
```

If a function is used as a part of an expression then it is important what goes after the function. If there brackets go next then the contents of these brackets is considered to be the arguments of this function. If there are no brackets right after the function then everything after the function name is considered to be the arguments of this function.

This example is quite clear:

```
a=SIN(x)+COS(y)
```

but without brackets

```
a=SIN x+COS y
```

it is equal to:

```
a=SIN(x+COS(y))
```

It is safe to put function arguments inside brackets always, but still it is you decide how to write:

```
PRINT(SIN(pi/4))
```

or

```
PRINT SIN(pi/4)
```

or

```
PRINT SIN pi/4
```

◇ ◇ ◇

Although many BASIC operators consist of several words, it is possible to write them in one word.

Such operators as:

```
GO TO, GO SUB, END IF, END DEF, RE DO, RE STORE
```

can be written as:

```
GOTO, GOSUB, ENDIF, ENDDEF, REDO, RESTORE
```

Many built-in functions consist of several words. All these functions can be written in one word, for example:

```
OPTION BASE 1
OPTIONBASE 1
DRAW CLEAR 0 0 0
DRAWCLEAR 0 0 0
```

◇ ◇ ◇

Several lines of code can be combined in a single line by using '`:`' lines separator.

These two code lines:

```
a = 5
PRINT a
```

can be written in a single line:

```
a = 5:PRINT a
```

◇ ◇ ◇

Single line can be split into several lines by using '`\`' line merging character.

These two code lines:

```
a b c d e f g h = \
1 2 3 4 5 6 7 8
```

represent a single line:

```
a b c d e f g h = 1 2 3 4 5 6 7 8
```

◇ ◇ ◇

Program code can include text from other files by specifying file name in '`{ }`' brackets.

The following code sample includes contents of file "libs.txt" in the middle:

```
a = 5
{libs.txt}
b = 10
```

Text inclusion using '`{ }`' brackets ensures that the specified text file will be included only once. All other text inclusions of the same file will be ignored. If it is

necessary to include the same file multiple times then '+' character should be used after first bracket: '{+ }'.

This code sample includes contents of file "addon/inc.txt" several times:

```
a = 5
{addon/inc.txt}
b = 10
{+addon/inc.txt}
```

File path of the including file will be calculated relatively to location of the source file. But it is possible to indicate file path relatively to the root folder by starting path with '/' character:

```
{/addon/inc.txt}
```

If file name contains spaces as in example below then it requires no special attention:

```
{more code.txt}
```

Comments

There are two types of comments: in-line comments and multi-line comments. In-line comment can comment only single line of code. Multi-line comment can comment any number of code lines.

In-line comment is set by `'` character. Text after `'` character and to the end of this line is a comment.

```
a = 5 'this is an in-line comment
```

Multi-line comment begins with '`[`' character and ends with '`]`' character. Text between '`[ ]`' brackets is a comment.

```
a = 5 [this is
a multi-line
comment]
```

Constants

Numeric constant can be written as an integer or a float value.

```
n = 10
z = 3.2
```

Numeric constant can be written in exponential notation.

```
y = 1.2E+2
```

Numeric constant can be written in hexadecimal notation.

```
x = #FF
w = #ffffff
```

Any number is a complex number. The imaginary part of a number is marked with a letter 'i' at the end.

```
n = 2+3i
```

◇ ◇ ◇

Text constant is a text inside " " quotes.

```
t = "this is a text"
```

Double quote character " inside a text string can be included by two adjacent double quotes characters "".

This code sample:

```
PRINT "This is a ""quoted"" word!"
```

outputs:

```
This is a "quoted" word!
```

Math operations

There are five math operations which can be used with numeric values:

- `+` plus
- `-` minus
- `*` multiply
- `/` divide
- `^` power

Example:

```
a = (2+3)*4
c = 2^8
```

◇ ◇ ◇

There is one string operation which can be used with text values:

- `+` plus

For example:

```
a = "my "
b = "text"
PRINT a+b
```

◇ ◇ ◇

If one of operands accepts the result of operation, for example:

```
a = a+1
b = b*(c-d)
```

then it can be written shorter:

```
a += 1
b *= c-d
```

Such combined assignment expressions are possible for all math and string operations:

```
+=
-=
*=
/=
^=
```

Arrays

There are three types of arrays: normal array (or just "array"), list and dictionary. As with variables, any element of any array has no fixed data type and can store numeric or text value.

Array is a multi-dimensional matrix of elements. Array can have any number of dimensions and any positive size in each dimension.

Array is declared by the `DIM` statement. The largest array indices in each dimension are specified in brackets. By default the first element in each dimension has index 0.

Examples of array definitions:

```
DIM m(20)
DIM v(10 20 30)
```

Single `DIM` statement can contain several array definitions:

```
DIM m(20) v(10 20 30)
```

It is possible to set initial array values when defining an array:

```
DIM a(3) = 5 10 "text" 1.5
```

If array is multi-dimensional then its initial values are listed as if its first dimension is iterated, then its second dimension is iterated, and so on:

```
DIM m(2 1) = m00 m10 m20,m01 m11 m21
```

Initial array values can be put in brackets, especially if there are more arrays defined next:

```
DIM a(3) = (5 10 "text" 1.5) m(20) v(10 20 30)
```

Array can be redefined later to another size or even to another type.

If array is redefined to the same dimension but to another size then its previous values are preserved whenever possible. This allows to change array size while keeping its existing values.

◇ ◇ ◇

List is a one-dimensional array with dynamic size and specific operations to manage its elements.

List is declared by `DIM` statement with empty brackets, because it has no predefined size:

```
DIM e()
```

By default newly created list is empty, but it is possible to specify initial list values exactly as it is done with arrays:

```
DIM e() = 5 10 "text" 1.5
```

List elements are accessed similarly to array elements.

The following code:

```
DIM v() = 10 20 30
PRINT v(0); v(1); v(2)
v(1) = v(0)+v(1)+v(2)
PRINT v(0); v(1); v(2)
```

outputs:

```
 10   20   30
 10   60   30
```

To add new values to a list the '+' operator is used. After the '+' operator a value to be added to the list should be specified. New value is added to the end of the list:

```
e + 2.5
e + "text"
```

The following code:

```
DIM v()
v + "value"
v + 25
PRINT v(0); v(1)
```

outputs:

```
value 25
```

To remove values from a list the '-' operator is used. After the '-' operator an index of an element to be removed from the list should be specified:

```
e - 4
```

The following code:

```
DIM v() = 10 20 30
PRINT v(0)
```

```
v - 0
PRINT v(0)
v - 0
PRINT v(0)
```

outputs:

```
 10
 20
 30
```

To insert a value into a list the '^' operator is used. After the '^' operator an index of an element before which empty element to be inserted should be specified:

```
e ^ 25
```

The following code:

```
DIM v()=10 20 30
PRINT v(2)
v ^ 0
PRINT v(2)
v ^ 0
PRINT v(2)
```

outputs:

```
 30
 20
 10
```

To trim a list the '/' operator is used. After the '/' operator an index of a list element to be trimmed should be specified:

```
e / 10
```

The following code:

```
DIM v() = 10 20 30
PRINT v(0); v(1); v(2)
v / 1
PRINT v(0); v(1); v(2)
```

outputs:

```
 10  20  30
 10  0  0
```

◇ ◇ ◇

Dictionary is a one-dimensional array with dynamic size and specific elements structure. Each element in a dictionary has its "key" and "value". "Key" is always a text string, "value" has no fixed data type and can store numeric or text value. All keys in a dictionary are unique. Dictionary keys are always sorted in ascending order.

Dictionary is declared by `DIM` statement with empty brackets, because it has no predefined size:

```
DIM e()
```

By default newly created dictionary is empty, but it is possible to specify initial dictionary pairs of keys and values by marking the end of each pair with '`;`' character:

```
DIM e() = "key1" 10;"key2" 20;
```

Value of an element in a dictionary is accessed by its key:

```
n = e("key")
```

The following code:

```
DIM v()
v("key1") = 10
v("key2") = 20
PRINT v("key1")+v("key2")
```

outputs:

```
 30
```

If to address a dictionary element using its index then it will return a key of this element.

The following code:

```
DIM v() = "key1" 10;"key2" 20;
PRINT v(0)
PRINT v(1)
```

outputs:

```
key1
key2
```

To remove values from a dictionary the '`-`' operator is used. After the '`-`' operator a key of an element to be removed from the dictionary should be specified:

```
e - "key1"
```

◇ ◇ ◇

By default the first element in an array has index 0. But it is possible to address the first element using index 1. This is done by using the function `OPTION BASE`:

`OPTION BASE 0` sets first element index to 0
`OPTION BASE 1` sets first element index to 1

The following code:

```
DIM m() = 1 2 3 4 5
PRINT m(1); m(2); m(3); m(4)
OPTION BASE 1
PRINT m(1); m(2); m(3); m(4)
```

outputs:

```
 2   3   4   5
 1   2   3   4
```

As `OPTION BASE` setting affects the first index number, this also affects the total number of elements in an array. Compare these two examples:

```
OPTION BASE 0
DIM m(2 2) = 1 2 3 4 5 6 7 8 9
```

and

```
OPTION BASE 1
DIM m(2 2) = 1 2 3 4
```

Labels and Jumps

There are two types of labels: number labels ans text labels.

Number labels are non-negative numbers in the beginning of code lines. They should be separated by space from the rest of code line if there is something after the label. These numbers can be not just integers but also floats:

```
10 a = 1
0.5 PRINT a
```

These number labels are never evaluated as numeric values and are always treated as strings. This means that labels "`10`", "`10.`" and "`10.0`" are three different labels.

◇ ◇ ◇

Text labels are words. Names of labels follow the same naming rules as variables. Any single word on a string which is not a function call is a text label:

```
label
a = 1
PRINT a
```

Several lines can be merged into one line using ':' lines separator, so previous code could look like this:

```
label:a = 1
PRINT a
```

◇ ◇ ◇

Jumping to a label is performed using `GO TO` operator:

```
GO TO label
```

This program:

```
GO TO 1
BACK:PRINT 2
GO TO OK
1 PRINT 1
GO TO BACK
OK:PRINT 3
```

outputs:

```
1
2
3
```

◇ ◇ ◇

Jumping to a label with return is performed using `GO SUB` and `RETURN` operators. The `GO SUB` operator jumps to a label, the `RETURN` operator returns back to continue program execution after `GO SUB` operator:

```
GO SUB label
...
label
...
RETURN
```

This code sample:

```
PRINT 1
GO SUB jump
PRINT 3
GO TO end
jump
PRINT 2
RETURN
end
PRINT 4
```

outputs:

```
1
2
3
4
```

◇ ◇ ◇

Indexed jumping to a label is performed using `ON GOTO` operator. Index of a label to be used for `GO TO` operatrion is specified after `ON` keyword:

```
ON k GOTO label1 label2 label3
```

Index of a label starts with 1. If index is less than 1 or greater than number of supplied labels then `GO TO` operator is ignored.

◇ ◇ ◇

Indexed jumping to a label with return is performed using `ON GOSUB` operator. Index of a label to be used for `GO SUB` operatrion is specified after `ON` keyword:

```
ON k GOSUB label1 label2 label3
```

Index of a label starts with 1. If index is less than 1 or greater than number of supplied labels then `GO SUB` operator is ignored.

Conditionals

There are two types of conditionals: single-line and multi-line conditionals.

Single-line conditional looks like this:

```
IF x THEN y ELSE z
```

where "x" is a condition, "y" is an operation if condition is true and "z" is an operation if condition is false. There can be only one operation after `THEN` and only one operation after `ELSE`.

The `ELSE` part is optional and can be omitted:

```
IF x THEN y
```

If operation is `GO TO`, for example:

```
IF x THEN GO TO label1
IF x THEN GO TO label1 ELSE GO TO label2
```

then `GO TO` operator can be omitted:

```
IF x THEN label1
IF x THEN label1 ELSE label2
```

◇ ◇ ◇

Multi-line conditional looks like this:

```
IF x THEN
...
ELSE IF y THEN
...
ELSE
...
END IF
```

First `IF` section is executed if "x" condition is true. Otherwise it goes to `ELSE IF` section which is executed if "y" condition is true. Otherwise it goes to `ELSE` section which is executed without any conditions. The end of a multi-line conditional is marked by the `END IF` statement. All these sections can contain any number of operations.

The `ELSE IF` section can be repeated as many times as needed:

```
IF x THEN
...
ELSE IF y THEN
...
ELSE IF z THEN
...
ELSE
...
END IF
```

The `ELSE IF` and `ELSE` sections are optional and can be omitted:

```
IF x THEN
...
END IF
```

The `THEN` keyword in a multi-line conditional is optional and can be omitted:

```
IF x
...
ELSE IF y
...
END IF
```

◇ ◇ ◇

Conditions can be true or false. False is 0. True is not false. This means that true is any number which is not zero.

There are following comparison operators:

`=` equal
`<` less
`>` greater
`<>` not equal
`<=` less or equal
`>=` greater or equal

Other variants of these operators: `><`, `=<`, `=>` are also valid.

There are four logic operators:

```
AND
OR
XOR
NOT
```

The `AND`, `OR` and `XOR` operators require left and right operands:

```
a AND b
```

The `NOT` operator requires only right operand:

```
NOT a
```

Priority of logic operators is less than of comparison operators.

All conditional expressions always evaluate as numbers: 0 for false and 1 for true.

This code sample:

```
PRINT (1<2)+(3=3)
```

outputs:

```
 2
```

User-defined functions

There are two types of user-defined functions: single-line and multi-line user-defined functions.

User function definition begins with `DEF` statement and then function name follows. After that optional list of function arguments can be specified.

Single-line user-defined function looks like this:

```
DEF f(x,y) = x+y
```

where "f" is a function name and "x" and "y" are function arguments. Function body goes after the '`=`' symbol.

Single-line function definition consists only of one code line and it allows to define a formula.

◇ ◇ ◇

Multi-line user-defined function looks like this:

```
DEF f(x,y)
...
END DEF
```

The end of a multi-line user function definition is marked with the `END DEF` statement. A multi-line user function can contain any number of code lines between `DEF` and `END DEF` statements.

Both multi-line and single-line user function definitions do not require putting function arguments in brackets. All function definitions below are exactly the same and it is up to you how to write:

```
DEF f(x,y)
DEF f(x y)
DEF f x,y
DEF f x y
```

◇ ◇ ◇

A multi-line user-defined function can return values using the `RETURN` statement. The `RETURN` statement can return any number of values or just finish function execution without returning any value.

Examples of `RETURN` statements:

```
RETURN
RETURN 1
RETURN a,b
```

Example of a user-defined function with the name "!" which calculates factorial of a number:

```
DEF ! x
IF x>1 THEN RETURN x*!(x-1) ELSE RETURN 1
END DEF

PRINT !(8)
```

and its output is:

```
 40320
```

◇ ◇ ◇

A user-defined function can accept arrays as function arguments and can return arrays as return values. Arrays are always submitted as their copies.

A user function can declare its own local arrays. Both local function arrays and those arrtays which were received as function argument can be returned from function as return values.

This example shows how the function "p" accepts an array as a function argument, modifies its first value and returns a modified array:

```
DEF p x
x(0) = 4
RETURN x
END DEF

DIM m() = 1 2 3
PRINT m(0); m(1); m(2)
m = p m
PRINT m(0); m(1); m(2)
```

and its output is:

```
 1  2  3
 4  2  3
```

This example shows how the function "p" creates an array and returns it as a function return value:

```
DEF p
DIM v() = 1 2 3
RETURN v
END DEF

DIM m()
PRINT m(0); m(1); m(2)
m = p
PRINT m(0); m(1); m(2)
```

and its output is:

```
 0   0   0
 1   2   3
```

◇ ◇ ◇

A user function definition can be located in any part of a program. If program execution reaches `DEF` statement of a function definition then it just skips this function definition to its `END DEF` statement and continues program execution. A user-defined function must be called explicitly to get executed.

A user-defined function can be executed by specifying its name. Using of brackets around function arguments is optional. Brackets around function arguments may be required if function call is a part of some expression.

These are examples of function calls:

```
f(x)
f x
func()
func
a = f(x)+f(y)
```

If a function returns more than one value then such function call must have a special mark, specifying how many values are expected to be returned here. This is achieved by using '`|`' character and a number of return values placed right after a function name:

```
f|2(x)
```

This sample program prints couples of values in an increasing order using the function "f" which takes two values and returns them back, lesser value first:

```
DEF f x,y
IF x<y THEN RETURN x,y ELSE RETURN y,x
```

```
END DEF

a b = f|2(2,3)
c d = f|2(4,1)
PRINT "a,b ="; a; b
PRINT "c,d ="; c; d
```

and it outputs:

```
a,b = 2  3
c,d = 1  4
```

◇ ◇ ◇

Any user-defined function is basically a subroutine with its own scope. All variables and labels which are used inside user-defined function belong exclusively to the function.

But it is possible to address any variable of any function from within any part of a program using a scope syntax: "variable scope"`.`"variable name", for example:

```
func.var
```

where `FUNC` is a name of the function and `VAR` is a name of the variable inside this function.

This sample program:

```
DEF f
a = 10
END DEF

f
PRINT f.a
```

outputs:

```
 10
```

because the function `PRINT` outputs a value of the variable A which belongs to the function F.

To address a global variable (variable which belongs to main program body) from within any function the same scope syntax should be used. But global variables have empty scope, so nothing should be specified before the dot character, for example:

```
.var
```

This sample program:

```
DEF f
PRINT .a
END DEF

a = 20
f
```

outputs:

```
 20
```

Cycles

There are two types of cycles: cycles by variable and cycles by condition.

A cycle by variable loops while specified variable changes its value from start value to stop value using incremental step value. A cycle by variable begins with the `FOR` statement and ends with the `NEXT` statement:

```
FOR i = 1 TO 10 STEP 1
  ...
NEXT
```

A loop variable is specified after the `FOR` statement, a start value is specified after the '`=`' symbol, a stop value is specified after the `TO` statement and a step value is specified after the `STEP` statement.

The `STEP` statement can be omitted if a step is equal to 1:

```
FOR i = 1 TO 10
  ...
NEXT
```

Start, stop and step values are defined only once when entering the cycle.

The `NEXT` statement redirects to the beginning of the cycle. The `NEXT` statement can contain any text after the keyword, this text has no effect and is considered to be just a comment:

```
FOR i = 1 TO 10
  ...
NEXT cycle
```

◇ ◇ ◇

A cycle by condition begins with the `DO` statement and ends with the `REDO` statement. The `DO` statement can have a condition to begin the loop. The `REDO` statement can have a condition to repeat the loop. Both `DO` and `REDO` conditions are optional. A condition is specified after the `IF` statement:

```
DO IF x<10
  ...
REDO
```

and

```
DO
   ...
REDO IF y<20
```

It is possible to use both conditions in the same cycle:

```
DO IF x<10
   ...
REDO IF y<20
```

If there is no condition then it means that it is met. The following cycle will repeat unlimitedly:

```
DO
   ...
REDO
```

◇ ◇ ◇

A cycle can be stopped at any time using the `BREAK` statement:

```
DO IF x<10
   ...
   IF y=20 THEN BREAK
   ...
REDO
```

The `BREAK` statement terminates the innermost cycle it belongs to. But it is possible to terminate outer loop by variable by specifying this loop variable after the `BREAK` statement:

```
FOR x = 0 TO 100
  FOR y = 0 TO 100
     ...
     IF y=n THEN BREAK X
     ...
  NEXT
NEXT
```

◇ ◇ ◇

A cycle can skip its current loop and proceed to its next loop using the `SKIP` statement:

```
DO IF x<10
   ...
   IF x=n THEN SKIP
```

```
    ...
  REDO
```

The `SKIP` statement skips the innermost cycle it belongs to. But it is possible to skip outer loop by variable by specifying this loop variable after the `SKIP` statement:

```
FOR x = 0 TO 100
  FOR y = 0 TO 100
    ...
    IF y=n THEN SKIP X
    ...
  NEXT
NEXT
```

Built-in data

Program can contain built-in data. Such data blocks begin with the `DATA` statement and can contain any number of any values and expressions. There can be any number of data blocks and they can be located in any part of the program:

```
...
DATA 1.5 "text" 2+3
...
DATA 10 20 30 0 0 0
```

◇ ◇ ◇

The `READ` operator performs sequential reading of built-in data into specified variables:

```
READ a,b
```

After each data reading the internal data pointer is increased, so next `READ` operator reads next data values.

The following code:

```
DATA 1 2 "Hello," "World"
READ a b
PRINT a; b
READ a b
PRINT a; b
```

outputs:

```
 1  2
Hello,World
```

◇ ◇ ◇

Built-in data statements are localized inside their scope. If some function has its own `DATA` statements then they are available for reading only from within this function.

This sample code:

```
DATA 1 2 3
f

DEF f
```

```
DATA 10 20 30
READ a b
PRINT a; b
END DEF
```

outputs:

```
 10  20
```

because the function F has its own `DATA`, so it ignores `DATA` in the main program body.

If some function has no `DATA` statements but has `READ` statements then it will read `DATA` from main program body.

This sample code:

```
DATA 1 2 3
f
DEF f
READ a b
PRINT a; b
END DEF
```

outputs:

```
 1  2
```

because the function F has no its own `DATA`, so it reads `DATA` from the main program body.

Similarly, `DATA` statements which are located inside functions are invisible for `READ` statements in main program body.

This sample code:

```
READ a b c d
PRINT a; b; c; d
DATA 1 2

DEF f
DATA 10 20 30 40
END DEF

DATA 3 4
```

outputs:

```
1   2   3   4
```

because the `DATA` inside the function F is ignored by the `READ` statement in the main program body.

This data localization by scope can be bypassed by `RESTORE TO` statement which is discussed later.

◇ ◇ ◇

The operator `RESTORE` rewinds data pointer to the beginning of all data blocks in the scope. So, next data reading will start from the beginning of data

The following code:

```
READ a b
PRINT a; b
RESTORE
READ c d
PRINT c; d
DATA 1,2
```

outputs:

```
1   2
```

◇ ◇ ◇

The operator `RESTORE TO` rewinds data pointer to the beginning of a data block which is located in the program after the specified label:

```
RESTORE TO label
```

The following code:

```
1 DATA 1,2
2 DATA 3,4
RESTORE TO 2
READ a b
PRINT a; b
```

outputs:

```
3   4
```

◇ ◇ ◇

In the `RESTORE TO` statement the label can be specified using scope syntax: "scope"`.`"label name". This means that it is possible to read data located in any scope from within any other scope.

This is an example of reading `DATA` inside a function from the main function body:

```
RESTORE TO f.1
READ a b
PRINT a; b


DEF f
1 DATA 10 20 30
ENDDEF
```

It outputs:

```
 10  20
```

This is an example of reading `DATA` located in the main function body from within a function which has its own `DATA`:

```
main:DATA 1 2 3
f


DEF f
DATA 10 20 30
RESTORE TO .main
READ a b
PRINT a; b
ENDDEF
```

It outputs:

```
 1  2
```

Strings

Strings consist of Unicode symbols. Diacritical marks are also parts of the symbols. Each symbol has a varied bytes count. That is why the length of a string measured in symbols is not the same as the size of the same string measured in bytes.

The following example:

```
PRINT STR LEN "José"
PRINT STR SIZE "José"
```

outputs:

```
4
13
```

where the first number is the length of the text in symbols and the second number is the size of the text in bytes.

All string processing functions operate in symbols. Of course it is possible to get a list of bytes from the string or to create a string from a list of separate bytes using such functions as `STR BYTES` or `STR TEXT`.

The following example:

```
PRINT STR TEXT #95A
```

outputs:

```
ĵ
```

Built-in functions

Console

`CLS`

Clears the text console.

`INPUT a[,b,...]`

Performs a request in the dedicated console input field to input values for the specified variables.

This function can have one or more arguments.

When typing several values separate them with commas.

Type of input values is detected automatically, but everything inside quotes is always accepted as a text value.

`OPTION TAB a[,b,...]`

Redefines the tabulation size in the `PRINT` function.

This function can have one or more arguments.

Each argument defines the size of current tabulation. Last argument also defines the size of all next tabulations.

`PRINT [a,b,...]`

Prints the specified arguments to the text console.

This is a special function because separators between arguments are also arguments. Comma separator '`,`' performs tabulation. Semicolon separator '`;`' allows to add more arguments.

Special `TAB(n)` subfunction moves printing position to the specified index. If current line length is equal or greater than this index then line feed is added to the end of current line. The position indices begin with 1.

```
PRINT TAB(n);a
```

A line feed is automatically added to the end of a string, but if the last argument is a separator then a line feed is not added.

```
PRINT a;
```

Without any arguments this function prints only a line feed.

Math

`ABS n`

Returns the absolute value of N.

`ACOS n`

Returns the arc cosine of N, in radians.

`ASIN n`

Returns the arc sine of N, in radians.

`ATAN n`

Returns the arc tangent of N, in radians.

`ATAN2 y,x`

Returns the arc tangent of Y/X, in radians.

This function takes into account the sign of both arguments in order to determine the quadrant.

`BITAND a,b`

Returns the result of bitwise AND operation between the arguments A and B.

`BITNOT n`

Returns the result of bitwise NOT operation for the argument N.

`BITOR a,b`

Returns the result of bitwise OR operation between the arguments A and B.

`BITXOR a,b`

Returns the result of bitwise XOR operation between the arguments A and B.

`CEIL n`

Returns the smallest integer value that is not less than N.

`COMPLEX a,b`

Returns a number which consists of the real part A and the imaginary part B.

`COS n`

Returns the cosine of N. Angle N should be specified in radians.

`EXP n`

Returns the constant e=2.71828... raised to the power of N.

`FLOOR n`

Returns the largest integer value that is not greater than N.

`IMAG n`

Returns the imaginary part of the number N.

`LOG n`

Returns the natural logarithm of N.

`LOG10 n`

Returns the common logarithm of N.

`MATH ARG n`

Returns the argument of the complex number N.

`MATH EVEN n`

Returns 1 if the number N is even, otherwise returns 0.

`MATH FFT m`

Returns as a list the result of a fast Fourier transform of the contents of the list or one-dimensional array M. The number of elements in M should be the integer power of 2.

`MATH IFFT m`

Returns as a list the result of an inverse fast Fourier transform of the contents of the list or one-dimensional array M. The number of elements in M should be the integer power of 2.

`MATH ODD n`

Returns 1 if the number N is odd, otherwise returns 0.

`MAX a,b`

Returns the maximum value of two numbers A and B.

`MIN a,b`

Returns the minimum value of two numbers A and B.

`MOD a,b`

Returns the floating point remainder of A/B (rounded towards zero).

`RANDOMIZE [n]`

Sets the random numbers sequence to N.

Argument N is optional, without it the random sequence is used.

`REAL n`

Returns the real part of the number N.

`RND`

Returns the random float number N, where 0 <= N < 1.

`RNDI x[,y]`

Returns the random integer number in the specified interval.

The second argument Y is optional.

With single argument X this function returns the number N, where 0 <= N <= X.

With two arguments X and Y this function returns the number N, where X <= N <= Y.

`ROUND n`

Returns the integer value that is nearest to N, with halfway cases rounded away from zero.

`SGN n`

Returns the sign of N: -1 if N < 0, 0 if N = 0 and 1 if N > 0.

`SIN n`

Returns the sine of N. Angle N should be specified in radians.

`SQR n`

Returns the square root of N. Value of N should not be negative.

`TAN n`

Returns the tangent of N. Angle N should be specified in radians.

Strings

`STR BYTES s`

Returns a list of bytes of the string S.

`STR CAPS s`

Returns the string S in the upper case. This affects only latin letters.

`STR CHAR m`

Returns a string consisting of specified bytes.

If M is a number then it is used as a Unicode character code.

If M is a list or one-dimensional array of bytes then it is used as a sequence of bytes in UTF-8 encoding.

`STR GUID`

Returns the GUID (globally unique identifier) string.

`STR HEIGHT s`

Returns a height in points of the string S if it is drawn with the current font settings.

`STR LEFT s,n`

Returns N left symbols of the string S.

`STR LEN s`

Returns a number of symbols in the string S.

`STR LOW s`

Returns the string S in the lower case. This affects only latin letters.

`STR MID s,b,n`

Returns a substring from the string S. The substring begins at position B and have a length of N symbols. The position numbering begins with 1.

`STR NUM s`

Returns a number converted from the string S.

This function supports all types of number formats available for numeric constants in a program code.

This function sets `ERROR` variable.

`STR POS s,ss`

Returns a position number of the substring SS in the string S. The position numbering begins with 1. If there is no occurrence of the substring SS in the string S then it returns 0.

`STR POSR s,ss`

Returns a position number of the substring SS in the string S if to search from the right. The position numbering begins with 1. If there is no occurrence of the substring SS in the string S then it returns 0.

`STR REP s,n`

Returns a string which is received by repeating N times the string S.

`STR RIGHT s,n`

Returns N right symbols of the string S.

`STR RPL s,ss,rs`

Returns a string S where each substring SS is replaced with string RS.

`STR SIZE s`

Returns a number of bytes in the string S.

`STR SPLIT s,c1[,c2,...]`

Returns a list of strings which are got by splitting the string S by specified separators.

`STR TEXT n[,f]`

Returns a string converted from the number N using optional format string F.

Format characters:

`?` - digit or space;
`0` - digit or zero;
`.` - decimal dot;
`E` - exponential notation (case sensitive);
`#` - hexadecimal digit.

Format string rules:

- all non-formatting characters go as is;
- format character '`E`' or '`e`' should be the last character.

Examples:

```
PRINT STR TEXT 1/3 ".??"
0.33

PRINT STR TEXT 1/3 ".??E"
3.33E-1
```

```
PRINT STR TEXT 10/4 ".00"
2.50

PRINT STR TEXT 255 "## ##"
00 FF

PRINT STR TEXT 1000000 "??? ??? ???"
  1 000 000

PRINT STR TEXT 123456789 "+0(000)000-00-00"
+0(012)345-67-89
```

`STR WIDTH s`

Returns a width in points of the string S if it is drawn with the current font settings.

Graphics

The top left corner of the screen has coordinates (0;0). The X-axis goes rightwards and the Y-axis goes downwards.

There is a difference between points and pixels. Points are device-independent screen measurement units. Pixels are hardware-dependent screen elements. Almost all drawing functions use points as arguments.

By default the screen is updated automatically. But it is possible to disable automatic screen update using `DRAW MANUAL` function and to update the screen manually using `DRAW UPDATE` function. Automatic screen update can be re-enabled using `DRAW AUTO` function.

All color functions operate with color values in range: 0..1.

◇ ◇ ◇

`DRAW AT x,y`

Sets current drawing pen position to coordinates (X;Y).

`DRAW AUTO`

Enables automatic screen update rate.

`DRAW CIRC x,y,r[,ry]`

Draws a circle with center at coordinates (X;Y) and radius R. If RY argument is specified then this function draws an ellipse with radius R in X-axis and radius RY in Y-axis.

`DRAW CLEAR r,g,b[,a]`

Clears graphics window into the color with red R, green G, blue B and optional alpha A color components. Components values N are 0 <= N <= 1.

`DRAW COLOR r,g,b[,a]`

Sets current drawing color to the color with red R, green G, blue B and optional alpha A color components. Components values N are 0 <= N <= 1.

`DRAW FCIRC x,y,r[,ry]`

Draws a filled circle with center at coordinates (X;Y) and radius R. If RY argument is specified then this function fills an ellipse with radius R in X-axis and radius RY in Y-axis.

`DRAW FONT [s][n]`

Sets the current font name to S and the current font size to N. Both S and N arguments are optional but at least one of them should be specified.

`DRAW FRECT x1,y1,x2,y2`

Draws a filled rectangle with one corner at point with coordinates (X1;Y1) and an opposite corner at point with coordinates (X2;Y2).

`DRAW IMAGE f,x,y[,s[,sy]]`

Draws an image from the file at path F placing its top left corner at coordinates (X;Y). If the argument S is specified then the image is drawn using scale S. If S and SY arguments are specified then the image is drawn using scale S in X-axis and scale SY in Y-axis.

`DRAW LINE x1,y1,x2,y2`

Draws a line from point with coordinates (X1;Y1) to point with coordinates (X2;Y2). Current drawing pen position is updated with this function.

`DRAW MANUAL`

Disables automatic screen update rate.

`DRAW PIXEL x,y[,x2,y2]`

If two arguments X and Y are specified then this function draws a pixel at coordinates (X;Y).

If four arguments are specified then this function draws a line of pixels from point at coordinates (X;Y) to point at coordinates (X2;Y2).

The coordinates are measured in pixels, not in points.

`DRAW POINT x,y`

Draws a point at coordinates (X;Y).

`DRAW RECT x1,y1,x2,y2`

Draws a rectangle with one corner at point with coordinates (X1;Y1) and an opposite corner at point with coordinates (X2;Y2).

`DRAW SIZE n`

Sets drawing pen size to value N.

`DRAW TEXT s,x,y[,a]`

Draws the text string S at point with coordinates (X;Y) at optional angle A. Angle is specified in radians and is measured in counter-clockwise direction. The string is drawn using the current font settings.

`DRAW TO x,y`

Draws a line from current drawing pen position to position with coordinates (X;Y). Current drawing pen position is updated with this function.

`DRAW UPDATE`

Performs manual screen update.

`HSV2RGB|3 h,s,v`

Converts a color from HSV color space to RGB color space and returns red, green and blue values which correspond to hue H, saturation S and value V.

`RGB2HSV|3 r,g,b`

Converts a color from RGB color space to HSV color space and returns hue, saturation and value which correspond to red R, green G and blue B color components.

`SCR CENTER|2`

Returns the X and Y coordinates of the graphics screen center, in points.

`SCR HEIGHT`

Returns the height of the graphics screen, in points.

`SCR SCALE`

Returns the pixel/point ratio of the graphics screen.

`SCR SIZE|2`

Returns the width and the height of the graphics screen, in points.

`SCR SIZE w,h`

Sets the width and the height of the graphics screen to the values W and H, in points.

`SCR WIDTH`

Returns the width of the graphics screen, in points.

Files

When working with files the file pointers are managed automatically. You always can check or change any file pointer position using the `FILE POS` function. All reading and writing functions are performed from the current file pointer position.

The result of the file reading function `FILE READ` depends on the current file reading mode. There are two file reading modes: `TEXT` and `BYTES`. The default file reading mode is `TEXT`.

When speaking about a file name it is implied that this name can be not just a file name but a file path. A file name is a text value.

◇ ◇ ◇

`DIR CREATE f`

Creates the directory F.

This function sets `ERROR` variable.

`DIR DELETE f`

Deletes the directory F.

`DIR DIRS f`

Returns a list of directories in directory F.

`DIR EXISTS f`

Returns 1 if the directory F exists. Otherwise returns 0.

`DIR FILES f`

Returns a list of files in directory F.

`DIR RENAME f,g`

Renames the directory with name F to new path G. If a directory at path G already exists then renaming will be not performed.

This function sets `ERROR` variable.

`DIR RUN`

Returns a directory from which the program was run.

`FILE BYTES`

Sets the file reading mode to `BYTES`.

`FILE DELETE f`

Deletes the file with name F.

`FILE EXISTS f`

Returns 1 if the file with name F exists. Otherwise returns 0.

`FILE POS f[,n]`

If argument N is absent then this function returns the current position of the file pointer for the file with name F.

If argument N is present then this function sets the current position of the file pointer to the value N for the file with name F.

`FILE READ f[,n]`

If the file reading mode is `TEXT` and if the argument N is absent then this function returns a text string with the contents of the file with name F.

If the file reading mode is `TEXT` and if the argument N is present then this function returns a text string with N symbols from the file with name F.

If the file reading mode is `BYTES` and if the argument N is absent then this function returns the contents of the file with name F as a list of bytes.

If the file reading mode is `BYTES` and if the argument N is present and if N = 1 then this function returns a single byte from the file with name F.

If the file reading mode is `BYTES` and if the argument N is present and if N > 1 then this function returns a list of N bytes from the file with name F.

This function sets `ERROR` variable.

`FILE RENAME f,g`

Renames the file with name F to new path G. If a file at path G already exists then renaming will be not performed. File pointer positions for the file names F and G will be reset.

This function sets `ERROR` variable.

`FILE SIZE f`

Returns the size (in bytes) of the file with name F.

`FILE TEXT`

Sets the file reading mode to `TEXT`.

`FILE TRUNC f,n`

Truncates the length of the file with name F to the specified value N (in bytes).

`FILE WRITE f,a[,b,...]`

Writes the specified arguments to the file with name F.

If an argument is a text value then it is written in UTF-8 encoding.

If an argument is a numeric value then it is written as a byte. Numeric value N should be 0 <= N <= 255.

If an argument is a list or one-dimensional array then the array elements are written as a sequence of bytes. All array elements should be numeric values. Each numeric value N should be 0 <= N <= 255.

This function sets `ERROR` variable.

Interface

Interface functions operate with interface objects. All interface objects are identified by object indices which are integer values: 0, 1, 2..., they can be even negative values. Each type of interface objects has its own set of object indices. A button with the object index 1 and a text field with the object index 1 are different objects.

Positions of all interface objects are measured relatively their top left corner.

An interface function which sets any parameter of a non-existing interface object creates this object automatically.

When working with touches each touch is destinguished by its number. The first touch gets number 0. If the first touch is still pressed and another touch is getting pressed then the new touch gets number 1, and so on. Each new touch gets the lowest number available, beginning with 0. Each touch is tracked individually and keeps its number while it is being pressed.

Mouse clicks are treated as touches, but they always have fixed touch numbers: left button click is a touch number 0, right button click is a touch number 1 and middle button click is a touch number 2.

◇ ◇ ◇

`INBUTT BCOLOR k,r,g,b[,a]`

Sets the background color to the color with red R, green G, blue B and optional alpha A color components for a button with the object index K.

`INBUTT BORDER k,n[,r,g,b[,a]]`

Sets the border thickness to the value N for a button with the object index K. Optionally this function sets the border color to the color with red R, green G, blue B and optional alpha A color components.

`INBUTT DELETE k`

Deletes a button with the object index K.

`INBUTT DISABLE k`

Disables a button with the object index K.

`INBUTT ENABLE k`

Enables a disabled button with the object index K.

`INBUTT FCOLOR k,r,g,b[,a]`

Sets the font color to the color with red R, green G, blue B and optional alpha A color components for a button with the object index K.

`INBUTT FONT k,[s][n]`

Sets the font name to S and the font size to N for a button with the object index K. Both S and N arguments are optional but at least one of them should be specified.

`INBUTT HIDE k`

Hides a button with the object index K.

`INBUTT HIT k`

Returns 1 if a button with the object index K was pressed. Otherwise it returns 0.

`INBUTT INDEX`

Returns next free object index for a button.

`INBUTT POS k,x,y`

Sets a position of a button with the object index K to the coordinates (X;Y).

`INBUTT SET k,t,x,y,w,h`

This function sets the text T, the position coordinates (X;Y), the width W and the height H for a button with the object index K.

`INBUTT SHOW k`

Shows a hidden button with the object index K.

`INBUTT SIZE k,w,h`

Sets the width W and the height H for a button with the object index K.

`INBUTT TEXT k,t`

Sets the text T for a button with the object index K.

`INEDIT CHANGED k`

Returns 1 if an edit field with the object index K has finished editing ('Enter' button is pressed or control has lost its focus). Otherwise it returns 0.

`INEDIT DELETE k`

Deletes an edit field with the object index K.

`INEDIT DISABLE k`

Disables an edit field with the object index K.

`INEDIT ENABLE k`

Enables a disabled edit field with the object index K.

`INEDIT HIDE k`

Hides an edit field with the object index K.

`INEDIT INDEX`

Returns next free object index for an edit field.

`INEDIT LOCK k`

Locks editing in an edit field with the object index K.

`INEDIT POS k,x,y`

Sets a position of an edit field with the object index K to the coordinates (X;Y).

`INEDIT SELECT k`

Selects a text in an edit field with the object index K.

`INEDIT SET k,t,x,y,w`

This function sets the text T, the position coordinates (X;Y) and the width W for an edit field with the object index K.

`INEDIT SHOW k`

Shows a hidden edit field with the object index K.

`INEDIT TEXT k[,t]`

If the second argument is present then this function sets the text T in an edit field with the object index K.

If the second argument is absent then this function returns the text in an edit field with the object index K.

`INEDIT UNLOCK k`

Unlocks editing in an edit field with the object index K.

`INEDIT WIDTH k,w`

Sets the width W for an edit field with the object index K.

`INSIGN ALIGN k,n`

Sets the text alignment for a sign field with the object index K.

N=1 - left alignment;

N=2 - right alignment;

N=3 - center alignment.

`INSIGN BCOLOR k,r,g,b[,a]`

Sets the background color to the color with red R, green G, blue B and optional alpha A color components for a sign field with the object index K.

`INSIGN DELETE k`

Deletes a sign field with the object index K.

`INSIGN FCOLOR k,r,g,b[,a]`

Sets the font color to the color with red R, green G, blue B and optional alpha A color components for a sign field with the object index K.

`INSIGN FONT k,[s][n]`

Sets the font name to S and the font size to N for a sign field with the object index K. Both S and N arguments are optional but at least one of them should be specified.

`INSIGN HIDE k`

Hides a sign field with the object index K.

`INSIGN INDEX`

Returns next free object index for a sign field.

`INSIGN POS k,x,y`

Sets a position of a sign field with the object index K to the coordinates (X;Y).

`INSIGN SET k,t,x,y,w,h`

This function sets the text T, the position coordinates (X;Y), the width W and the height H for a sign field with the object index K.

`INSIGN SHOW k`

Shows a hidden sign field with the object index K.

`INSIGN SIZE k,w,h`

Sets the width W and the height H for a sign field with the object index K.

`INSIGN TEXT k,t`

Sets the text T for a sign field with the object index K.

`TOUCH HIT n`

Returns 1 if the touch with number N is currently pressed, otherwise returns 0.

`TOUCH POS|2 n`

Returns the position coordinates (X;Y) of the touch with number N.

Miscellaneous

`CON SHOW`

Switches to the console window.

`CPU RELAX`

Reduces the program execution speed. Very effective in the awaiting loops which do not require the fastest performance to reduce the CPU consumption.

`DELAY n`

Pauses the program execution for N seconds.

`END`

Stops the program execution.

`LABEL EXISTS s`

Returns 1 if a label with the name S exists, otherwise returns 0.

`NOW|3`

Returns the current hour, minute and second.

`OPTION BASE n`

Sets the lowest index of an array, it can be 0 or 1.

`SCR SHOW`

Switches to the graphics window.

`SIZE OF m[,n]`

If M is a list or a dictionary then this function returns the number of elements in the list or dictionary M.

If M is an array then it is important if there is one or two arguments of the function. If there is only one argument then this function returns the dimension of the array M. If there are two arguments then this function returns the number of elements in the array M in its N-th dimension, N >= 1.

`SORT DOWN m[,n]`

Returns as a list the result of sorting of M in descending order.

If M is a list or one-dimensional array then this function returns the sorted contents of M.

If M is a dictionary then this function returns the keys of the sorted values of M.

If M is a two-dimensional array then this function returns the indices of the sorted values of the column N of the array M.

`SORT UP m[,n]`

Returns as a list the result of sorting of M in ascending order.

If M is a list or one-dimensional array then this function returns the sorted contents of M.

If M is a dictionary then this function returns the keys of the sorted values of M.

If M is a two-dimensional array then this function returns the indices of the sorted values of the column N of the array M.

`TIMER GET`

Returns current timer value, in seconds. The `CPU RELAX` function affects the timer precision if used in the same loop.

`TIMER SET`

Resets timer value to 0.

`TODAY|3`

Returns the current year, month (1..12) and day (1..31).

`TYPE OF n`

Returns a number which describes the type of the object N:

1 = variable containing number

2 = variable containing text

3 = fixed-sized array

4 = list

5 = dictionary

`WEEKDAY y,m,d`

Returns a day of the week for the given date: year Y, month M and day D.

Monday = 1, ..., Sunday = 7.

Built-in constants

`2PI`

Mathematical constant 2*π=6.28318...

`CR$`

Carriage return character (#0D).

`LF$`

Line feed character (#0A).

`PI`

Mathematical constant π=3.14159...

`PI2`

Mathematical constant π/2=1.57079...

`TAB$`

Tabulation character (#09).

System variables

`ERROR`

Some functions affect this system variable. They set its value to indicate the correctness of the function execution. You also can modify this system variable, for example to reset it.

0 = no error
1 = file/directory access error
2 = amount of read/written data is not what was expected
3 = text is not a number

Function equivalents

The following function names are also valid and they are equivalent to the corresponding functions. It is up to you which function name to use:

```
ATN = ATAN
INT = FLOOR
SQRT = SQR
STOP = END
```

Appendices

Deviations from Minimal BASIC standard

In Minimal BASIC the `DIM` and the `OPTION BASE` statements can be jumped over but still have their effect.

In BestBASIC the `DIM` and the `OPTION BASE` statements should be explicitly executed to have their effect.

◇ ◇ ◇

In Minimal BASIC the `DATA` statement can use text values without quotes.

In BestBASIC all text values in the `DATA` statement should be put in quotes.

◇ ◇ ◇

In Minimal BASIC the `INPUT` statement performs data type conversion according to the variables types.

In BestBASIC there are no variables types, so the type of input data is autodetected. If numbers need to be treated as text they should be put in quotes.