RCALC embedded language and its runtime compiler

Mariusz M. Nogala

24 June 2016

RCALC is a small and simple **embedded language** (similar to C) and a **runtime compiler** for Windows programmers.
It allows you to execute a dynamic script code from the level of your own application **in the runtime**.

RCALC module is a part of **REPCoder.DLL** (**REPCoder64.DLL**).
It has the programming interfaces for C, C++, C#, Java, Delphi.

www.repcoder.com

Table of Contents

1. About RCALC ... 6
2. The structure of a script ... 6
3. Data types ... 7
4. Variable names ... 7
5. Text constants and date literals ... 7
6. Numbers ... 8
7. Keywords ... 9
8. INPUT section ... 9
9. Local variables section ... 9
10. OUTPUT section ... 10
11. Mathematical formulas ... 10
12. IF, ELSEIF, ELSE, WHILE ... 12
13. ROW_NUMBER, ROW_COUNT, ROW_TYPE ... 13
14. Type conversions ... 13
15. Built-in functions ... 14
16. Comments ... 15
17. Simple examples ... 16
    1) Add two integers ... 16
    2) Add two texts without truncation ... 16
    3) Add two texts and truncate the output value to 20 characters ... 16
    4) Local variables and library functions: sqrt(), random() ... 17
    5) IF, ELSEIF, ELSE control block ... 17
    6) Add texts using a WHILE loop ... 17
    7) Expand a text much faster using the += operator ... 18
    8) Write your own functions and use it in the main script ... 19
    9) Empty INPUT and OUTPUT sections ... 19
    10) The NULL keyword and functions: cond(), isnull(), isnotnull(), ifnull() ... 20
    11) WARNING() and totext() functions ... 20
    12) Operations on date, time, timestamp ... 21
    13) NewLine character and function substr() ... 21
    14) Vector parameters and function getfield() ... 22
    15) Global variables – using SETPARAM() and GETPARAM() ... 22
    16) Functions: min(), max() ... 24
    17) Query a database ... 24
    18) Using tables ... 25
18. Internal RCALC library functions ... 26
    1) ??? cond(INT Condition, ??? Value1, ??? Value2) ... 27
    2) NUMERIC round(NUMERIC Value, INT Precision) ... 27
    3) NUMERIC roundmoney(NUMERIC Value, INT Precision, INT Method) ... 27
    4) NUMERIC abs(NUMERIC Value) ... 27
    5) ??? max(??? Value1, ??? Value2) ... 27
    6) ??? min(??? Value1, ??? Value2) ... 27
    7) INT neg(INT Condition) ... 28
    8) NUMERIC mod(NUMERIC x, NUMERIC y) ... 28
    9) INT imod(INT x, INT y) ... 28
    10) NUMERIC sqrt(NUMERIC x) ... 28
    11) TEXT upper(TEXT t) ... 28
    12) TEXT lower(TEXT t) ... 28
    13) INT len(TEXT t) ... 28
    14) TEXT substr(TEXT s, INT First, INT nChars, INT FromEnd) ... 29
    15) INT char(TEXT s, INT n) ... 29
    16) INT ansi_char(TEXT s, INT n) ... 29
    17) INT findchar(TEXT s, INT Char) ... 29
    18) INT findchar2(TEXT s, INT Start, INT Char) ... 29
    19) INT findstr(TEXT s1, TEXT s2) ... 29
    20) INT findstr2(TEXT s1, INT Start, TEXT s2) ... 30
    21) TEXT add(TEXT s1, TEXT s2) ... 30

22) TEXT garbage_out(TEXT s, TEXT garbage) 30
23) TEXT replace(TEXT s, TEXT s1, TEXT s2) 30
24) INT strlike(TEXT s, TEXT format) 30
25) TEXT trim(TEXT s) 31
26) TEXT ltrim(TEXT s) 31
27) TEXT rtrim(TEXT s) 31
28) TEXT blank(INT n) 31
29) TEXT blank2(INT n, INT c) 31
30) INT writetext(TEXT fname, INT append, TEXT s) 31
31) TEXT readtext(TEXT fname, INT start, INT n) 31
32) INT writetext2(TEXT fname, INT append, TEXT s, INT format) 32
33) TEXT readtext2(TEXT fname, INT start, INT n, INT format) 32
34) INT day(DATE/TIMESTAMP d) 32
35) INT month(DATE/TIMESTAMP d) 32
36) INT year(DATE/TIMESTAMP d) 32
37) INT monthdays(DATE/TIMESTAMP d) 33
38) INT monthdays2(INT month, INT year) 33
39) DATE currdate() 33
40) DATE date(INT day, INT month, INT year) 33
41) INT hour(TIME/TIMESTAMP t) 33
42) INT minute(TIME/TIMESTAMP t) 33
43) INT second(TIME/TIMESTAMP t) 33
44) INT millisecond(TIME/TIMESTAMP t) 33
45) TIME currtime() 33
46) TIMESTAMP currtimestamp() 34
47) TIME time(INT hour, INT minute, INT second, INT millisec) 34
48) TIMESTAMP timestamp(INT day, INT month, INT year, INT hour, INT minute, INT second, INT millisec) 34
49) DATE getdate(TIMESTAMP tm) 34
50) TIME gettime(TIMESTAMP tm) 34
51) INT isnull(??? Value) 34
52) INT isnotnull(??? Value) 34
53) ??? ifnull(??? Value1, ??? Value2) 34
54) DATE todate(TEXT t) 35
55) TIME totime(TEXT t) 35
56) TIMESTAMP totimestamp(TEXT t) 35
57) TEXT totext(NUMERIC Value) 36
58) TEXT totext2(NUMERIC Value, INT Precision) 36
59) TEXT tochar(INT code) 36
60) TEXT moneytotext(NUMERIC Value) 36
61) TEXT moneytowords(NUMERIC Value, INT Language, INT Currency) 36
62) TEXT moneytowords2(NUMERIC Value, INT Language, TEXT Curr1, TEXT Curr2) 36
63) TEXT datetotext(DATE/TIME/TIMESTAMP d, INT format) 37
64) NUMERIC tonumber(TEXT t) 37
65) INT toint(NUMERIC x) 37
66) INT tobool(INT x) 37
67) NUMERIC binand(NUMERIC x1, NUMERIC x2) 37
68) NUMERIC binor(NUMERIC x1, NUMERIC x2) 37
69) NUMERIC binneg(NUMERIC x) 38
70) TEXT GETPARAM(INT n) 38
71) INT SETPARAM(INT n, TEXT Value) 38
72) INT PARAMFIELDCOUNT(INT n) 38
73) TEXT GETPARAMFIELD(INT n, INT f) 38
74) INT SETPARAMFIELD(INT n, INT f, TEXT Value) 38
75) INT ADDPARAMFIELD(INT n, TEXT Value) 38
76) INT DROPPARAMFIELD(INT n, INT f) 38
77) INT WARNING(TEXT message) 38
78) INT QUESTION(TEXT question) 39
79) INT STATUSINFO(TEXT message) 39

80) INT DB_CONNECT(INT method, TEXT dbname, TEXT user, TEXT pass, TEXT role, TEXT charset, INT memocp, INT writemode, INT show_error_message) 39
81) INT DB_DISCONNECT(INT dbnum) 40
82) INT SET_CONNECT_OPT(INT dbnum, INT method, TEXT dbname, TEXT user, TEXT pass, TEXT role, TEXT charset, INT memocp, INT writemode) 40
83) INT EXECSQL(TEXT sql) 40
84) INT EXECSQL2(INT dbnum, TEXT sql, INT show_error_message) 41
85) INT QUERY_LOAD(INT dbnum, TEXT sql, INT show_error_message) 41
86) INT QUERY_FREE(INT query_handle) 41
87) INT QUERY_COLS(INT query_handle) 42
88) INT QUERY_ROWS(INT query_handle) 42
89) TEXT QUERY_COLNAME(INT query_handle, INT colnr) 42
90) INT QUERY_COLTYPE(INT query_handle, INT colnr) 42
91) TEXT QUERY_COLTYPENAME(INT query_handle, INT colnr) 42
92) TEXT QUERY_RESULT(INT query_handle, INT rownum, INT colnr) 43
93) TEXT QUERY_RESULT2(INT query_handle, INT rownum, TEXT colname) 43
94) TEXT SELECT_SINGLE(INT dbnum, TEXT sql, INT show_error_message) 43
95) TEXT getfield(TEXT row, int fieldnr) 44
96) INT getfieldcount(TEXT row) 44
97) INT TAB_ALLOC(INT nitems, INT item_size) 44
98) INT TAB_REALLOC(INT tabnum, INT new_nitems) 45
99) INT TAB_FREE(INT tabnum) 45
100) NUMERIC TAB_GETITEM(INT tabnum, INT pos, INT numeric_scale) 45
101) TEXT TAB_GETTEXTITEM(INT tabnum, INT pos) 45
102) INT TAB_SETITEM(INT tabnum, INT pos, ??? value) 45
103) INT TAB_SET(INT tabnum, INT pos, INT nitems, ??? value) 46
104) INT TAB_APPEND(INT tabnum, ??? value) 46
105) INT TAB_INSERT(INT tabnum, INT pos, ??? value) 46
106) INT TAB_APPEND2(INT tabnum1, INT tabnum2, INT pos2, INT nitems) 46
107) INT TAB_INSERT2(INT tab1, INT pos1, INT tab2, INT pos2, INT nitems) 46
108) INT TAB_REMOVE(INT tabnum, INT pos, INT nitems) 46
109) INT TAB_GETCOUNT(INT tabnum) 47
110) INT TAB_GET_ITEMSIZE(INT tabnum) 47
111) INT TAB_CHANGE_ITEMSIZE(INT tabnum, INT new_item_size) 47
112) INT TAB_COPY(INT tab1, INT pos1, INT tab2, INT pos2, INT nitems) 47
113) INT TAB_READ(INT tabnum, INT pos, INT nitems, TEXT fname, INT fpos) 47
114) INT TAB_WRITE(INT tab, INT pos, INT nitems, TEXT fname, INT append) 47
115) INT TAB_FROMFILE(TEXT fname) 48
116) INT TAB_TOFILE(INT tabnum, TEXT fname) 48
117) INT PUT_BLOB(INT dbnum, TEXT TableName, TEXT FieldName, TEXT WhereSql, INT tabnum, INT show_error_message) 48
118) INT PUT_MEMO(INT dbnum, TEXT TableName, TEXT FieldName, TEXT WhereSql, TEXT Memo, INT show_error_message) 48
119) TEXT currdir() 48
120) INT random(INT n) 49
121) INT file_len(TEXT fname) 49
122) INT file_exists(TEXT fname) 49
123) INT DELETE_FILE(TEXT fname) 49
124) TEXT get_odbc_drivers() 49
125) TEXT find_files(TEXT filter) 49
126) INT convert_image(INT src_tabnum, INT image_type, INT jpg_quality) 50
19. Exported functions of the RCALC API 50
20. A sample application in C 63
21. A sample application in C++ 64
22. A sample application in C# 67
23. A sample application in Java 70
24. A sample application in Delphi 72
25. The RCALC compiler and thread-safety 75
26. A multi-thread example in C 76

27. A multi-thread example in C++ 80
28. A multi-thread example in C# 82

1. About RCALC

At its origin RCALC was created as an internal script language (similar to C) for our reporting tool **REPCoder** (www.repcoder.com). In a report definition it makes it possible to define output **calculated fields** as functions of those returned by SQL query. This mechanism extends the report applications enabling complex processing and transformation of data.

SQL query fields (INPUT) → RCALC → Calculated fields (OUTPUT)

Recently RCALC was also enabled for programmers of Windows applications in the form of an exported module in **REPCoder.DLL**. The RCALC API is provided for languages: C, C++, C#, Delphi, Java. Generally - for any language that can call exported functions of a standard Windows DLL.

Using this API the final application can execute at runtime a dynamic script code, passing necessary input parameters to it. The text of an algorithm can be read from any possible source (external files, databases, network, edit windows) as required by the host program.

Possible applications of the RCALC module:

- to calculate and produce some output values for the calling program
- to read and write files
- to connect with databases and execute SQL queries

RCALC is a small and compact embedded language. It implements the NUMERIC data type based on 64-bit integers. There are also internal DATE, TIME and TIMESTAMP types. The NULL value is also implemented, because it often happens in the database world. It also has control blocks like IF, ELSEIF, ELSE and the WHILE loop. The internal library now has more than 120 built-in functions. Users can also define their own local functions that can be used in the final (main) algorithm. In the current version the language controls data types **very strictly**. So it is often necessary to use the built-in conversion functions.

2. The structure of a script

The basic structure of the RCALC script is very simple:

```
INPUT {
<input section>
}

<local variables section>

OUTPUT {
<output section>
}
```

The section of local variables is optional. INPUT and OUTPUT are necessary, but can be empty.

3. Data types

Data types in RCALC:

- **FLOAT** double precision, 64-bit
- **INT** 32-bit signed integer
- **DATE** 32-bit integer
- **TIME** 32-bit integer
- **TIMESTAMP** 64-bit integer
- **TEXT** UNICODE text
- **NUMERIC (scale)** 64-bit signed integer

Remarks:

1) The TEXT type is using the Windows UNICODE format (2 bytes per character).
2) TEXT variables are dynamic and unlimited.
3) In the OUTPUT section it is allowed to specify the length of a text output type.
4) The scale for the NUMERIC type is the number of decimal digits, following the decimal dot. The minimum scale is 0. The maximum scale is 18. The scale is optional. The default value of scale is 0 (64-bit integer type).

So the numeric types are: NUMERIC (0), NUMERIC (1), ... , NUMERIC (18).

4. Variable names

It is possible to put variable names in quotation marks. It is however necessary if they contain spaces or other reserved characters. Variable names cannot have apostrophes or other quotation marks inside.

Important:
RCALC is a case-sensitive language (for type, variable and function names).

5. Text constants and date literals

Text constants have to be placed in apostrophe characters. They can have any possible characters inside (even quotation marks and other apostrophes). However the apostrophe characters must be preceded with the reserved character: \ (example: 'abcd\'efgh').

Important:

1) UNICODE codes of characters (INT type) have the following format: '\A', '\a', '\B', '\b'

```
INT codea = '\a';
INT codeA = '\A';
```

2) To get a new line character, we must use the built-in function tochar() and add the two results:

```
TEXT NewLine = tochar(13) + tochar(10);
```

3) To get a one-character text containing only a single apostrophe, we must also use the built-in function tochar():

```
TEXT Ap = tochar('\'');
```

4) To use a text as a date literal we must precede it with the reserved character: \.

```
DATE d1 = '\15-04-2002';
DATE d2 = '\15.04.2002';
DATE d3 = '\15/04/2002';
DATE d4 = '\2002-04-15';
DATE d5 = '\2002.04.15';
DATE d6 = '\2002/04/15';
```

Only the above 6 date formats are recognized.

6. Numbers

All number literals (with or without the decimal dot) are translated to the NUMERIC type (with some scale) and stored in memory as 64-bit integers. The idea is to perform any possible arithmetic operations as exact as possible. Only in necessary cases (if it cannot be avoided) there is a conversion to the double precision floating point type (FLOAT).

Examples:

```
10        NUMERIC (0)
9.45      NUMERIC (2)
3.14159   NUMERIC (5)
```

When we add or subtract, the scale of the result is the maximum scale of the two operands:

```
9.45 + 3.14159 = 12.59159          NUMERIC (5)
```

When we multiply or divide, the scale of the result is the sum of scales of the two operands:

```
9.45 * 3.14159 = 29.6880255        NUMERIC (7)
```

7. Keywords

The keywords (case sensitive) used in RCALC:

IF, ELSEIF, ELSE	control blocks
WHILE, BREAK, CONTINUE	loop
NULL	NULL value for FLOAT, INT, NUMERIC
NULL_DATE	NULL value for DATE
NULL_TIME	NULL value for TIME
NULL_TIMESTAMP	NULL value for TIMESTAMP

Extended keywords (used and documented in REPCoder calculated fields):

- **ROW_NUMBER**

predefined constant name of the NUMERIC(0) type, that holds the current row number of the SQL query result set

- **ROW_COUNT**

predefined constant name of the NUMERIC(0) type, that holds the number of rows in the result set of a sorted SQL query (with order by)

- **ROW_TYPE**

predefined variable name of the INT type, that returns a row type (-1, 0, 1, 2, 3, ...)

8. INPUT section

In this section we define the set of the input variables for the algorithm. Their types and names may correspond with the fields returned by the SQL query. The declarations must be placed in the block { ... }. Each instruction must end with a semicolon. It is possible to declare a few variables of the same type in a single instruction.

```
INPUT {
FLOAT          "Some percent";
INT            "Number of days", "Salary group";
TEXT           "Surname";
NUMERIC(2)     "Salary";
}
```

Important:
In the INPUT section for TEXT variables we do not specify the text length parameter.

9. Local variables section

This section allows to define and calculate local variables needed to obtain the final values in the OUTPUT section. Also some necessary job can be done there on files or databases. This section is placed just between the INPUT and OUTPUT sections. It is not marked in any other way. Only this section can have control blocks (keywords: IF, ELSEIF, ELSE) and loops (keywords: WHILE, BREAK, CONTINUE). The TEXT local variables are declared without the length parameter. They are fully dynamic without a limit of a number of characters.

There are 6 types of instructions, that can be placed in the local section:

1). `TYPE_NAME VariableName = <defining formula>;`
Define a new local variable (always together with its value).

2). `VariableName = <defining formula>;`
Set a new value of a local variable defined earlier.

3). IF, ELSEIF, ELSE control blocks

4). WHILE loops

5). Direct calls of special functions: SETPARAM, WARNING, STATUSINFO, EXECSQL, EXECSQL2, SETPARAMFIELD, ADDPARAMFIELD, DROPPARAMFIELD.
These special functions do not require to return value to any local or output variable.

6). Single-line comments (in C++ style: //).

10. OUTPUT section

Here we define the final calculated fields for the calling application. The definitions must be placed in the block { ... }. Each of them must be ended with the semicolon and have the following format:

```
TYPE_NAME FieldName = <defining formula>;
```

Important:

A defining formula can use:
1) input variables (from the INPUT section)
2) local variables (from the local section)
3) output variables already defined earlier.

11. Mathematical formulas

The mathematical formulas can use:

- **brackets:** ()
- **arithmetic operators:** +, -, *, /, ^ (power)

- **increment operator:** +=
- **logical operators:** || (logical OR), && (logical AND)
- **comparison operators:** >, >=, <, <=, ==, !=
- **standard add texts operator:** +
- **fast expand text operator:** +=
- **text and numeric constants, date and time literals**
- **variable names defined earlier**
- **built-in RCALC functions**
- **user functions defined earlier**
- **NULL, NULL_DATE, NULL_TIME, NULL_TIMESTAMP keywords**
- **markers of the report parameters: %1, %2, … (REPCoder only)**

Example:

```
FLOAT z = ((x + y)/2 + 10)*round(k, 2) - 3*y*y;
INT i_empty = NULL;
DATE dt_empty = NULL_DATE;
TIME tm_empty = NULL_TIME;
TIME tms_empty = NULL_TIMESTAMP;
TEXT t_empty = '';
NUMERIC(2) n = 1.23;
TEXT t_local = 'abcdefg' + 'hijkl';
INT len1 = len(t_empty + t_local);
DATE tomorrow = currdate() + 1;
INT rnd = random(1000);
TEXT t_output(7) = 'abcdefg';
```

Important:

1) If we define a calculated field of TEXT type in the OUTPUT section, we can also specify the length of the text. If we don't do it, the length will be unlimited in RCALC. But in REPCoder it will obtain the default value: 255 characters, because the size of calculated report fields must be exactly defined.

2) You cannot set with NULL variables of the TEXT type. In this case you should use the empty text constant – the 2 apostrophe characters: `''`. Variables of other types can be set with NULL.

3) Texts can be added with the + operator. However if you need to expand text you should use the much faster += operator (because it is using the realloc() memory routine). They are the only possible arithmetic operations on texts.

4) You can add or subtract an INT number (as a number of days) to a DATE variable. This way you can shift the date by the given number of days.

5) You can also add or subtract an INT number (as a number of seconds) to a TIME variable.

6) If you add or subtract a FLOAT number to a TIMESTAMP variable, it is also shifted according to the SQL standard. For example:

```
TIMESTAMP tm1 = '\2016-05-01 6:05:00';
```

```
TIMESTAMP tm2 = tm1 + 0.5;
TIMESTAMP tm3 = tm1 + 1;
// tm2 will be equal to: '2016-05-01 18:05:00'
// tm3 will be equal to: '2016-05-02 6:05:00'
```

12. IF, ELSEIF, ELSE, WHILE

A standard control block has the following format:

```
IF( <condition_1> )
  {
   <instructions>
  }
ELSEIF( <condition_2> )
  {
   <instructions>
  }

...

ELSEIF( <condition_n> )
  {
   <instructions>
  }
ELSE
  {
   <instructions>
  }
```

A while loop has the following format:

```
WHILE( <condition> )
  {
   <instructions>
  }
```

Important:

1) ELSEIF and ELSE blocks are optional.

2) The body of a block must be placed in { ... }. It is required even for a single instruction block.

3) Inside a control block or a loop other blocks and loops can be nested.

4) You cannot define any new local variable inside a block or a loop. You can only set the values of existing local variables already defined earlier.

5) Inside the WHILE loop you can use BREAK and CONTINUE instructions (ended with semicolon).

6) Inside a block or a loop you can call directly special functions (without assigning their return values to any variables):
SETPARAM, WARNING, STATUSINFO, EXECSQL, EXECSQL2, SETPARAMFIELD, ADDPARAMFIELD, DROPPARAMFIELD

13. ROW_NUMBER, ROW_COUNT, ROW_TYPE

Special keywords used only in the REPCoder calculated fields:

ROW_NUMBER
This is a predefined constant of the NUMERIC(0) type, that holds the current row number of the SQL query result.

ROW_COUNT
This is a predefined constant of the NUMERIC(0) type, that holds the total number of rows in the SQL query result, or the number of rows in a group when the query is sorted with ORDER BY. It is used inside the aggregate rows (ROW_TYPE: -1, 1,2,3,...).

ROW_TYPE
This is a predefined constant of the INT type, that holds the type of a row:
0 standard rows
-1 the total aggregate row of the SQL query
1, 2, 3, … group aggregate rows of a sorted SQL query
(this is the aggregate rank - position in the ORDER BY clause).

Important:

While fetching the result set of an internal query, REPCoder is calculating the aggregate rows, that correspond to the aggregate SQL language functions: SUM, MIN, MAX. It always takes place for queries with the ORDER BY clause. They are so-called the grouping aggregate rows. For a query without ORDER BY only the total aggregate row is calculated (for the entire result set). The SUM aggregations are calculated only for number fields - INT, FLOAT, NUMERIC types. The MIN and MAX aggregations are calculated for number types and also for DATE, TIME, TIMESTAMP types.

14. Type conversions

All possible conversions between the numeric types are allowed: FLOAT, INT, NUMERIC. You cannot convert TEXT type to any numeric type and vice versa. If you convert a TEXT(n1) type to other TEXT(n2) type but a shorter one (n2<n1), the possible loss of trailing characters can occur. It is also not allowed to convert between DATE, TIME, TIMSTAMP and other types. But it is possible to use the RCALC conversion functions between TEXT and other types.

15. Built-in functions

The detailed description of the all (more than 120) built-in RCALC functions can be found in next chapters. Here is the listing only:

- **conditional-expression function:**

`cond`

- **rounding functions:**

`round, roundmoney`

- **logical negation:**

`neg`

- **some common functions:**

`abs, max, min, mod, imod, sqrt`

- **text functions:**

`len, upper, lower, substr, char, ansi_char, findchar, findchar2, findstr, findstr2, add, garbage_out, replace, strlike, trim, ltrim, rtrim, blank, blank2`

- **file read-write functions:**

`readtext, readtext2, writetext, writetext2`

- **date functions:**

`date, day, month, year, currdate, monthdays, monthdays2`

- **time, timestamp functions:**

`time, timestamp, hour, minute, second, millisecond, currtime, currtimestamp, getdate, gettime`

- **functions with NULL:**

`isnull, isnotnull, ifnull`

- **conversion functions:**

`todate, totime, totimestamp, totext, totext2, tochar, moneytotext, moneytowords, moneytowords2, datetotext, tonumber, toint, tobool`

- **binary functions:**

`binand, binor, binneg`

- **report parameters functions:**

`GETPARAM, SETPARAM, PARAMFIELDCOUNT, GETPARAMFIELD, SETPARAMFIELD, ADDPARAMFIELD, DROPPARAMFIELD`

- **MessageBox functions:**

`WARNING, QUESTION`

- **status function:**

`STATUSINFO`

- **database functions:**

```
DB_CONNECT, DB_DISCONNECT, SET_CONNECT_OPT,
EXECSQL, EXECSQL2,
QUERY_LOAD, QUERY_FREE,
QUERY_COLS, QUERY_ROWS,
QUERY_COLNAME, QUERY_COLTYPE, QUERY_COLTYPENAME,
QUERY_RESULT, QUERY_RESULT2,
SELECT_SINGLE, getfield, getfieldcount
```

- **memory tables functions:**

```
TAB_ALLOC, TAB_REALLOC, TAB_FREE,
TAB_GETITEM, TAB_GETTEXTITEM, TAB_SETITEM,
TAB_APPEND, TAB_APPEND2, TAB_INSERT, TAB_INSERT2,
TAB_REMOVE, TAB_SET, TAB_GETCOUNT, TAB_GET_ITEMSIZE,
TAB_CHANGE_ITEMSIZE, TAB_COPY, TAB_READ, TAB_WRITE,
TAB_FROMFILE, TAB_TOFILE
```

- **blob functions:**

```
PUT_BLOB, PUT_MEMO
```

- **utility functions:**

```
currdir, random, file_len, file_exists, DELETE_FILE, find_files,
get_odbc_drivers, convert_image
```

Important:

1) Short conditional expressions can be realized with the cond**()** function. It corresponds to the conditional expression in C/C++ or the iif() function in Microsoft Access.

2) There is no operator of a logical negation. Use the neg() function instead.

3) If you want to check if a value is NULL or not you should use the function isnull() or isnotnull().

4) Texts can be added with the operator + or by the add() function. To expand a text you should use the much faster += operator:

```
TEXT s = 'a'; s += 'b'; s += 'c';
```

16. Comments

In the current version comments are always in a single, separate line:

```
// This is a comment ...
```

Important:
Comments can be written only in the local section. Each comment must have its own line. It is not possible to put a comment in the same line following the RCALC code:

```
INT x = 1; // this is not allowed in RCALC
```

17. Simple examples

1) Add two integers

```
INPUT {
INT x;
INT y;
}

OUTPUT {
INT z = x + y;
}
```

2) Add two texts without truncation

```
INPUT {
TEXT x;
TEXT y;
}

OUTPUT {
TEXT z = x + y;
INT L = len(z);
}
```

Remarks:
TEXT is the output type for an unlimited length calculated automatically and not truncated. Function len() is an internal RCALC library function.

3) Add two texts and truncate the output value to 20 characters

```
INPUT {
TEXT x;
TEXT y;
}

OUTPUT {
TEXT(20) z = x + y;
}
```

Remarks:
TEXT(20) is the output text type with the maximum length of 20 characters. So the result will be truncated if it is too long.

4) Local variables and library functions: sqrt(), random()

```
INPUT {
FLOAT x;
FLOAT y;
}

FLOAT x2 = x*x;
FLOAT y2 = y*y;
FLOAT R2 = x2 + y2;
FLOAT z = (4*x2 + 2*x + 3)/(1 + y2*y + 0.5*(x+y)) + sqrt(R2);

OUTPUT {
FLOAT Out = z*random(100);
}
```

5) IF, ELSEIF, ELSE control block

```
INPUT {
INT x;
}

TEXT s = '';

IF( x < 10 )
  {
   s = 'x is < 10';
  }
ELSEIF( x < 100 )
  {
   s = 'x is < 100';
  }
ELSEIF( x < 1000 )
  {
   s = 'x is < 1000';
  }
ELSE
  {
   s = 'x is >= 1000';
  }

OUTPUT {
TEXT Out = s;
}
```

6) Add texts using a WHILE loop

```
INPUT {
```

```
}

INT n = 1000;
INT i = 0;
TEXT s = '';
WHILE( i < n )
  {
   s = s + 'a';
   i = i + 1;
  }

// write s to a file (0 - overwrite the existing file)
INT ok = writetext('a1000.txt', 0, s);

OUTPUT {
TEXT Out = s;
INT "Length of Out" = len(Out);
}
```

Remarks:
The INPUT section here is empty (no input values are required). The writetext() is not a special function, so it must return its value to some (local or output) variable. The second output value has a long name with spaces and must be placed in quotation marks "". A comment is used in the local section.

7) Expand a text much faster using the += operator

```
INPUT {
}

INT n = 1000000;
INT i = 0;
TEXT s = '';
WHILE( i < n )
  {
   s += 'a';
   i += 1;
  }

OUTPUT {
INT "Length of s" = len(s);
}
```

Remarks:
The += increment operator works much faster for texts. RCALC is using the C-language realloc() memory routine in this case. So it works in the same way as the += operator for C++ class std::string. The result is created in a few seconds. If you would use the standard + operator to expand s 1000000 times, it would last very long (about 30 minutes).
Here we have also used the += operator to increment the value of an integer i. But it works exactly the same as `i=i+1` (not faster).

8) Write your own functions and use it in the main script

```
INPUT {
INPUT {
FLOAT x;
FLOAT y;
}

OUTPUT {
FLOAT myfun1 = sqrt(x*x + y*y);
}

//--- This was a function: myfun1

INPUT {
}

DATE dt = currdate() + 1;

OUTPUT {
TEXT myfun2 = datetotext(dt, 6);
}

//--- This was a function: myfun2

INPUT {
FLOAT x;
FLOAT y;
}

OUTPUT {
FLOAT Radius = myfun1(x, y);
TEXT Tomorrow = myfun2();
}
```

Remarks:
A user function is coded in the same way as the main algorithm. It must also have the INPUT (can be empty) and OUTPUT sections. The only difference is that the OUTPUT section must have exactly one output value. Its name becomes just the name of the function. Only the last algorithm in the script can have many OUTPUT values and it works as the main function in standard programming languages. The INPUT and OUTPUT sections are always necessary and they can be empty in the main algorithm.

9) Empty INPUT and OUTPUT sections

```
INPUT {
}

TEXT s = '';
INT i = 0;
WHILE( i < 100 )
  {
```

```
   IF( random(2) == 1 ) { s += 'a'; }
   ELSE { s += 'b'; }
   i += 1;
  }

INT ok = writetext('ab100.txt', 0, s);

OUTPUT {
}
```

Remarks:
The aim of this script is only to produce and write a file with a randomly generated text.

10) The NULL keyword and functions: cond(), isnull(), isnotnull(), ifnull()

```
INPUT {
INT x;
INT y;
}

INT a = NULL;
INT z1 = x + a;
INT z2 = y * a;

IF( isnull(z2) )
  {
   STATUSINFO('\nNULL z2 ...\n');
  }

z2 = ifnull(z2, 0);

TEXT s1 = cond(isnull(z1), 'z1 is null', 'z1 is not null');
TEXT s2 = cond(isnotnull(z2), 'z2 is not null', 'z2 is null');

OUTPUT {
INT Out_z1 = z1;
INT Out_z2 = z2;
TEXT Out_s1 = s1;
TEXT Out_s2 = s2;
}
```

Remarks:
The NULL value quickly propagates with every arithmetic operation. The cond() function works as the conditional expression. The STATUSINFO() function by default writes to the program Console, but it can be redirected to any GUI Window handle (using the rcalc_set_status_window() RCALC API function).

11) WARNING() and totext() functions

```
INPUT {
NUMERIC(2) x;
}
```

```
WARNING('You have enetered: ' + totext(x));

OUTPUT {
TEXT Out = totext2(x, 4);
}
```

12) Operations on date, time, timestamp

```
INPUT {
}
DATE dt_now = currdate();
TIME tm_now = currtime();
TIMESTAMP tms_now = currtimestamp();

// New line to Console
STATUSINFO('');

DATE dt2 = getdate(tms_now);
IF( dt2 == dt_now )
  {
   STATUSINFO('EQUAL dates ...');
  }

TIME tm2 = gettime(tms_now);
IF( tm2 == tm_now )
  {
   STATUSINFO('EQUAL times ...');
  }

STATUSINFO('Day = ' + totext(day(dt_now)) + ', Month = ' +
totext(month(dt_now)) + ', Year = ' + totext(year(dt_now)));

TEXT s = 'Month days = ' + totext(monthdays(dt_now));
s += ', Today is: ' + datetotext(tms_now, 6);
STATUSINFO(s);

OUTPUT {
DATE out_dt = dt_now;
DATE out_dt_plus_1 = dt_now + 1;
TIME out_tm = tm_now;
TIME out_tm_plus_1 = tm_now + 1;
TIMESTAMP out_tms = tms_now;
TIMESTAMP out_tms_plus_1 = tms_now + 1;
TIMESTAMP "out_tms_plus_0.5" = tms_now + 0.5;
DATE out_battle_at_grunwald = date(15, 7, 1410);
TIMESTAMP tm3 = '\2016-05-01 6:05:00';
TIMESTAMP tm4 = tm3 + 0.5;
TIMESTAMP tm5 = tm3 + 1;
}
```

13) NewLine character and function **substr()**

```
INPUT {
}
TEXT NewLine = tochar(13) + tochar(10);
TEXT s = 'ABCDEFGHIJK';

OUTPUT {
TEXT substr_from_begin = substr(s, 2, 3, 0);
TEXT substr_from_end = substr(s, 2, 3, 1);
TEXT MultiLine = s + NewLine + s + NewLine + s + NewLine + s;
}
```

Remarks:
The function tochar() returns a single character text of the specified 2-byte code (UNICODE).

14) Vector parameters and function getfield()

```
INPUT {
}

// This text has 6 fields separated with ###
TEXT s = 'Johny###was born###1990-10-17###and has###3###children';

INT nFields = getfieldcount(s);
INT i = 1;

WHILE(i <= nFields)
  {
   STATUSINFO('Field nr ' + totext(i) + ': ' + getfield(s, i));
   i += 1;
  }

OUTPUT {
DATE dt = todate(getfield(s, 3));
INT n = tonumber(getfield(s, 5));
INT nFields1 = getfieldcount('');
INT nFields2 = getfieldcount('###');
INT nFields3 = getfieldcount('######');
}
```

Remarks:
In REPCoder and RCALC you can define a vector text composed of a number of fields. The fields are separated with the ### sequence inside the host text. You can get a given field calling the getfield() function with its 1-based number as an argument. The number of fields is returned by the function getfieldcount(). This useful practice is often used in REPCoder where you can define report complex parameters in the form of such vectors. In this way a single parameter can store multiple values (for example a whole record of SQL result).

15) Global variables – using SETPARAM() and GETPARAM()

```
INPUT {
```

```
}

OUTPUT {
INT Fun1 = tonumber(GETPARAM(1));
}
//---------------------

INPUT {
}

OUTPUT {
FLOAT Fun2 = tonumber(GETPARAM(2));
}
//---------------------

INPUT {
}

OUTPUT {
TEXT Fun3 = GETPARAM(3);
}
//---------------------

INPUT {
}

OUTPUT {
TEXT Fun4 = GETPARAM(100);
}
//---------------------

INPUT {
INT n;
FLOAT d;
TEXT s;
}

SETPARAM(1, totext(n));
SETPARAM(2, totext(d));
SETPARAM(3, s);

OUTPUT {
INT out_n = Fun1();
FLOAT out_d = Fun2();
TEXT out_s = Fun3();
TEXT out_empty = Fun4();
}
```

Remarks:

In the current version of RCALC there are no special global variables that you can declare between the function definitions and then use inside the functions. However to obtain the same behavior you can use the SETPARAM() and GETPARAM() built-in library functions. The argument passed to them is the 1-based integer (as the parameter's number). Parameters are just hidden global text variables in RCALC. You access a parameter through its number. The function SETPARAM(n, 'value') creates a parameter and GETPARAM(n) returns its

value as the TEXT type. There is no limitation on the length of the value. If the n-th parameter was not set before, calling GETPARAM(n) will return an empty text. You do not need to worry about the sequence of parameter numbers. Calling SETPARAM(100, 'value') is quite legal even if parameters with lower numbers were not set. You should also note that SETPARAM() is a kind of a special function in RCALC, that can be called directly without returning value to any (local or output) variable.
The idea of parameters as global variables comes from REPCoder where the report parameters are just global texts. They can be set and get in many places inside the report's code. The parameter numbers play exactly the same role as the global variable names.

16) Functions: min(), max()

```
INPUT {
}

INT m = 1 + random(12);
INT y = random(2030);
INT d = 1 + random(monthdays2(m, y));

OUTPUT {
DATE RandDate = date(d,m,y);
}
//-----------------

INPUT {
}

OUTPUT {
DATE dt1 = RandDate();
DATE dt2 = RandDate();
DATE dt_min = min(dt1, dt2);
DATE dt_max = max(dt1, dt2);
}
```

Remarks:
The min() and max() functions work for any RCALC type, not only numbers.

17) Query a database

```
INPUT {
}

INT n = NULL;
TEXT res = '';
TEXT connstr = 'DRIVER={Microsoft Text Driver (*.txt; *.csv)}';
TEXT sql1 = 'select count(*) from "employee.csv"';
TEXT sql2 = 'select * from "employee.csv" where "Id" = 5';
INT db = DB_CONNECT(2, connstr, '', '', '', '', 0, 0, 1);

IF( db > 1000 )
  {
```

```
   STATUSINFO('DB_CONNECT Succes ...');
   res = SELECT_SINGLE(db, sql1, 1);
   n = tonumber(res);
   res = SELECT_SINGLE(db, sql2, 1);
   DB_DISCONNECT(db);
  }
ELSE
  {
   STATUSINFO('DB_CONNECT Error !!!');
  }

OUTPUT {
INT "Number of employees" = n;
TEXT Row5 = res;
TEXT LastName = getfield(res, 2);
TEXT FirstName = getfield(res, 3);
DATE BirthDate = todate(getfield(res, 4));
NUMERIC(2) Salary = tonumber(getfield(res, 7));
}
```

Remarks:
In this example we use the function DB_CONECT() to connect with the "employee.csv" file using the ODBC Microsoft Text Driver. Then we use the function SELECT_SINGLE() to execute and get the first result row of some SQL select-type query. There are 2 sql queries there: sql1, sql2. The result Row5 of the sql2 query contains fields separated with the ### sequence. To extract a given field we use the getfield() function.

18) Using tables

```
INPUT {
INT tab;
TEXT fname;
}

NUMERIC(0) sum = NULL;
INT n = TAB_GETCOUNT(tab);

TEXT NL = tochar(13) + tochar(10);
INT i = 0;
INT item = 0;
TEXT savetxt = '';
IF( n > 0 )
  {
   sum = 0;
   WHILE( i < n )
     {
      item = TAB_GETITEM(tab, i, 0);
      sum += item;
      savetxt += totext(item) + NL;
      i += 1;
     }
  }
```

```
INT ok = writetext(fname, 0, savetxt);

IF( ok == 1 )
  {
   STATUSINFO('Table was saved to the file: ' + fname + ' ...' +
NL);
  }

OUTPUT {
NUMERIC(0) SumAndSaveTable = sum;
}
//---------------------

INPUT {
}

INT n = 1000;
INT tabnum = TAB_ALLOC(n, 8);

INT i = 0;
WHILE( i < n )
  {
   TAB_SETITEM(tabnum, i, random(30000));
   i += 1;
  }

OUTPUT {
NUMERIC(0) SumTable = SumAndSaveTable(tabnum, 'tabfile.txt');
}
```

Remarks:
In this example we use the function TAB_ALLOC() to allocate a numeric table of 1000 integers. Each table item has the size equal 8 bytes (64-bit integers). The table is filled with random numbers. Then it is passed to the user function SumAndSaveTable() which sums the item values and writes the table to a text file.

18. Internal RCALC library functions

In the current version there are 126 built-in functions in the RCALC language library. They are integrated with the language and are not to be declared in the source code of a script. Their declarations are listed below.

Important:
As it was already mentioned RCALC is a language with a strict control of variable types. However there are functions that accept arguments or return values of any RCALC type (or for example numbers only). If a function input argument can be of any possible type, it is marked as ???.

1) ??? **cond**(INT Condition, ??? Value1, ??? Value2)

The conditional-expression function.

Return value:
If argument 1 (Condition) is true (non zero): Value1
else: Value2

2) NUMERIC **round**(NUMERIC Value, INT Precision)

Standard rounding function of a floating number Value (up or down).
Precision: number of output decimal digits

3) NUMERIC **roundmoney**(NUMERIC Value, INT Precision, INT Method)

Extended rounding function of a monetary Value.

Precision: number of decimal digits
Method:
0 - standard (>= 0.5 - to up, < 0.5 - to down)
1 - to up
-1 - to down

4) NUMERIC **abs**(NUMERIC Value)

Absolute value of a number.

5) ??? **max**(??? Value1, ??? Value2)

Returns maximum of two values.

Return value:
If Value1 > Value2: Value1
else: Value2

6) ??? **min**(??? Value1, ??? Value2)

Returns minimum of two values.

Return value:
If Value1 < Value2: Value1
else: Value2

7) INT **neg**(INT Condition)

Negation of a logical expression.

8) NUMERIC **mod**(NUMERIC x, NUMERIC y)

Returns x modulo y.

9) INT **imod**(INT x, INT y)

Returns x modulo y of two integers.

10) NUMERIC **sqrt**(NUMERIC x)

Returns square root of x.

11) TEXT **upper**(TEXT t)

Returns text with uppercase characters.

12) TEXT **lower**(TEXT t)

Returns text with lowercase characters.

13) INT **len**(TEXT t)

Returns the length of text (number of characters).

14) TEXT **substr**(TEXT s, INT First, INT nChars, INT FromEnd)

Returns the substring of a text s.

First: Number of the first character.

nChars: Maximum number of characters to copy.
if < 0 - all next characters are copied.

FromEnd:
0 - from the beginning
1 - from the end

15) INT **char**(TEXT s, INT n)

Returns integer code (UNICODE) of the n-th character of a text s.

16) INT **ansi_char**(TEXT s, INT n)

Returns ANSI code (converted from UNICODE) of the n-th character of a text s.

17) INT **findchar**(TEXT s, INT Char)

Scans the text s for the first occurrence of the character Char (UNICODE).

Returns the number of the found character.
If it is not found, returns 0.

18) INT **findchar2**(TEXT s, INT Start, INT Char)

Scans the text s for the first occurrence of the character Char (UNICODE).
The scanning begins from the Start character.

Returns the number of the found character.
If it is not found, returns 0.

19) INT **findstr**(TEXT s1, TEXT s2)

Scans the text s1 for the first occurrence of a substring s2.

Returns the number of the first character of the found substring.
If it is not found, returns 0.

20) INT **findstr2**(TEXT s1, INT Start, TEXT s2)

Scans the text s1 for the first occurrence of a substring s2.
The scanning begins from the Start character.

Returns the number of the first character of the found substring.
If it is not found, returns 0.

21) TEXT **add**(TEXT s1, TEXT s2)

Joins two texts s1, s2.

Important:
Texts (two or more) can be also easily joined by the standard + operator.

22) TEXT **garbage_out**(TEXT s, TEXT garbage)

Cuts out all garbage characters from the source text s.
Returns the copy of the cleaned text s.

23) TEXT **replace**(TEXT s, TEXT s1, TEXT s2)

Replaces all texts s1 with the text s2 in the source text s.
Returns the changed copy of the text s.

24) INT **strlike**(TEXT s, TEXT format)

Checks if the text s has a given format.
Returns logical 1 or 0.

Examples:

```
strlike('America', '*ca') = 1
strlike('America', '??e*') = 1
strlike('America', '*co*') = 0
```

25) TEXT **trim**(TEXT s)

Cuts out the leading and trailing spaces.

26) TEXT **ltrim**(TEXT s)

Cuts out the leading spaces.

27) TEXT **rtrim**(TEXT s)

Cuts out the trailing spaces.

28) TEXT **blank**(INT n)

Returns empty string containing n space characters.

29) TEXT **blank2**(INT n, INT c)

Returns a string containing n equal characters c (UNICODE).

30) INT **writetext**(TEXT fname, INT append, TEXT s)

Writes the text s to the file fname in the ANSI format. It overwrites the existing file (if append=0) or appends the text to the end of file (if append=1). The file is opened before and closed after the operation.

Returns: 1 (success), 0 (error).

31) TEXT **readtext**(TEXT fname, INT start, INT n)

Reads n characters from the file fname in the ANSI format starting from the start position (1-based). If n = NULL or n <= 0 the entire file is read.

Returns the read text.

32) INT **writetext2**(TEXT fname, INT append, TEXT s, INT format)

Writes the text s to the file fname in the specified format. It overwrites the existing file (if append=0) or appends the text to the end of file (if append=1). The file is opened before and closed after the operation.

The format parameter can have the following values:
0- ANSI, 1- UTF8, 2- UTF8 (+BOM), 3- Unicode, 4- Unicode reversed.
Its value can also be any codepage number used by WideCharToMultiByte() Windows API function.

Important:
The BOM marker is written to the beginning of file only if the append parameter is NULL or 0.

Returns: 1 (success), 0 (error).

33) TEXT **readtext2**(TEXT fname, INT start, INT n, INT format)

Reads n characters from the file fname in the specified format starting from the start position (1-based). If n = NULL or n <= 0 the entire file is read.

The format parameter can have the following values:
0- ANSI, 1- UTF8, 2- UTF8 (+BOM), 3- Unicode, 4- Unicode reversed.
Its value can also be any codepage number used by MultiByteToWideChar() Windows API function.

Important:
The function skips the BOM marker if present.

Returns the read text.

34) INT **day**(DATE/TIMESTAMP d)

Returns month day for DATE or TIMESTAMP d.

35) INT **month**(DATE/TIMESTAMP d)

Returns month number for DATE or TIMESTAMP d.

36) INT **year**(DATE/TIMESTAMP d)

Returns year day for DATE or TIMESTAMP d.

37) INT **monthdays**(DATE/TIMESTAMP d)

Returns the number of month days for DATE or TIMESTAMP d.

38) INT **monthdays2**(INT month, INT year)

Returns the number of month days for given: month, year.

39) DATE **currdate**()

Returns the current date.

40) DATE **date**(INT day, INT month, INT year)

Returns date for given: day, month, year.

41) INT **hour**(TIME/TIMESTAMP t)

Returns hour for the TIME or TIMESTAMP variable t.

42) INT **minute**(TIME/TIMESTAMP t)

Returns minute for the TIME or TIMESTAMP variable t.

43) INT **second**(TIME/TIMESTAMP t)

Returns second for the TIME or TIMESTAMP variable t.

44) INT **millisecond**(TIME/TIMESTAMP t)

Returns millisecond for the TIME or TIMESTAMP variable t.

45) TIME **currtime**()

Returns the current time.

46) TIMESTAMP **currtimestamp**()

Returns the current timestamp.

47) TIME **time**(INT hour, INT minute, INT second, INT millisec)

Returns time for given: hour, minute, second, millisec.

48) TIMESTAMP **timestamp**(INT day, INT month, INT year, INT hour, INT minute, INT second, INT millisec)

Returns timestamp for given: day, month, year, hour, minute, second, millisec.

49) DATE **getdate**(TIMESTAMP tm)

Extracts DATE part for given TIMESTAMP tm.

50) TIME **gettime**(TIMESTAMP tm)

Extracts TIME part for given TIMESTAMP tm.

51) INT **isnull**(??? Value)

Checks if the value is null.
Returns logical 1 (null) or 0 (not null).

Important:
An empty TEXT is interpreted as null (the function returns 1).

52) INT **isnotnull**(??? Value)

Checks if the value is not null.
Returns logical 1 (not null) or 0 (null).

Important:
An empty TEXT is interpreted as null (the function returns 0).

53) ??? **ifnull**(??? Value1, ??? Value2)

Returns:
If Value1 is null: Value2 else: Value1

54) **DATE todate**(TEXT t)

Conversion function of text to date.
Returns date value if the text has one of the 6 formats:

'dd.mm.rrrr', 'dd-mm-rrrr', 'dd/mm/rrrr'
'rrrr.mm.dd', 'rrrr-mm-dd', 'rrrr/mm/dd'

Otherwise the null value is returned.

Important:
If we want to use text as a date literal we must precede it with the '\' character.

Examples:
```
DATE d1 = '\15-04-2002';
DATE d2 = '\15.04.2002';
DATE d3 = '\15/04/2002';
DATE d4 = '\2002-04-15';
DATE d5 = '\2002.04.15';
DATE d6 = '\2002/04/15';
```

55) TIME **totime**(TEXT t)

Conversion function of text to TIME.
Returns TIME value if the text has the time format:

h:m:s.ms (example: 14:24:02.198)

Otherwise the null value is returned.

Important:
If we want to use text as a TIME literal we must precede it with the '\' character.

Examples:
```
TIME t1 = '\14:24';
TIME t2 = '\14:24:02';
TIME t3 = '\14:24:02.198';
```

56) TIMESTAMP **totimestamp**(TEXT t)

Conversion function of text to TIMESTAMP.
Returns TIMESTAMP value if the text has one of the 6 DATE formats
(together with space-seperated time format: TM = h:m:s.ms):

```
'dd.mm.rrrr TM', 'dd-mm-rrrr TM', 'dd/mm/rrrr TM'
'rrrr.mm.dd TM', 'rrrr-mm-dd TM', 'rrrr/mm/dd TM'
```

Otherwise the null value is returned.

Important:
If we want to use text as a TIMESTAMP literal we must precede it with the '\' character.

Examples:
```
TIMESTAMP t1 = '\15-04-2002';
TIMESTAMP t2 = '\15.04.2002 14:24';
TIMESTAMP t3 = '\15/04/2002 14:24:02';
TIMESTAMP t4 = '\2002-04-15 14:24:02.198';
```

57) TEXT **totext**(NUMERIC Value)

Simple conversion function of a number to text.

58) TEXT **totext2**(NUMERIC Value, INT Precision)

Extended conversion function of a number to text.

Precision: number of decimal digits.
If Precision = -1 or NULL the full (default) precision is returned.

59) TEXT **tochar**(INT code)

Conversion function of a code (UNICODE) to a single character text.

60) TEXT **moneytotext**(NUMERIC Value)

Conversion function of a number to text using the Windows money display format.

61) TEXT **moneytowords**(NUMERIC Value, INT Language, INT Currency)

Conversion function of a number to text in words using the given language and currency.

Language: 0 (default), 1, 2, 3, ...
Currency for the language: 0 (default), 1, 2, 3, ...

REPCoder:
The available display languages together with their currencies are defined in the Report Properties dialog.

62) TEXT **moneytowords2**(NUMERIC Value, INT Language, TEXT Curr1, TEXT Curr2)

Conversion function of a number to text in words using the given language and currency symbols.

Language: 0 (default), 1, 2, 3, ...
Currency symbols: Curr1 (major), Curr2 (minor)

REPCoder:
The available display languages are defined in the Report Properties dialog

63) TEXT **datetotext**(DATE/TIME/TIMESTAMP d, INT format)

Conversion function of DATE/TIME/TIMESTAMP to text using the specified DATE format:

```
1: dd/mm/yyyy
2: dd.mm.yyyy
3: dd-mm-yyyy
4: yyyy/mm/dd
5: yyyy.mm.dd
6: yyyy-mm-dd
7: dd/mm/yy
8: mm/dd/yy
```

For TIME variable the format argument is ignored.

64) NUMERIC **tonumber**(TEXT t)

Conversion function of a text to number. If the text cannot be converted it returns NULL.

65) INT **toint**(NUMERIC x)

Conversion function of a floating point number to the integer value. It cuts out the fractional part.

66) INT **tobool**(INT x)

Conversion function of a number to the logical value 1 or 0.
If the given argument is non-zero returns 1, otherwise returns 0.

67) NUMERIC binand(NUMERIC x1, NUMERIC x2)

Returns binary AND of two 64-bit values x1, x2.

68) NUMERIC **binor**(NUMERIC x1, NUMERIC x2)

Returns binary OR of two 64-bit values x1, x2.

69) NUMERIC **binneg**(NUMERIC x)

Returns binary negation of a 64-bit value x.

70) TEXT **GETPARAM**(INT n)

Returns the value of the n-th parameter of the report (REPCoder) or a global text (RCALC) in the runtime.

71) INT **SETPARAM**(INT n, TEXT Value)

Sets the value of the n-th parameter of the report (REPCoder) or a global text in the runtime.
Returns: 1 (success), 0 (bad number n).

72) INT **PARAMFIELDCOUNT**(INT n)

Returns the number of fields of the n-th parameter in the runtime.

73) TEXT **GETPARAMFIELD**(INT n, INT f)

Returns the value of the field number f (1-based) for the n-th parameter in the runtime.

74) INT **SETPARAMFIELD**(INT n, INT f, TEXT Value)

Sets the value of the field number f (1-based) for the n-th parameter in the runtime.
Returns: 1 (success), 0 (bad number n or f).

75) INT **ADDPARAMFIELD**(INT n, TEXT Value)

Adds a new field for the n-th parameter in the runtime.
Returns: 1 (success), 0 (bad number n).

76) INT **DROPPARAMFIELD**(INT n, INT f)

Drops the field f for the n-th parameter in the runtime.
Returns: 1 (success), 0 (bad number n or f).

77) INT **WARNING**(TEXT message)

Displays a warning message box in the runtime.
Returns: 1 (success), 0 (error).

Important:
To insert a new line inside the message use: \n or |n

78) INT **QUESTION**(TEXT question)

Displays a question MessageBox for the user in the runtime.
Returns: 1 (user response = YES), 0 (user response = NO).

Important:
To insert a new line inside the question text use: \n or |n

79) INT **STATUSINFO**(TEXT message)

Displays a message text in the status window in the runtime.
REPCoder: It is located in the left-upper corner of the REPCoder main window.
RCALC: It can be set to any Window handle. The default zero value represents Console.

Returns: 1 (success), 0 (error).

80) INT **DB_CONNECT**(INT method, TEXT dbname, TEXT user, TEXT pass, TEXT role, TEXT charset, INT memocp, INT writemode, INT show_error_message)

Creates a database connection.

Arguments:

method:
(1-BDE, 2-ODBC, 3-Firebird, 4-Interbase)

dbname:
1 (BDE):
BDE alias (e.g. "ALIAS: BCDEMOS") or BDE directory (for *.DB, *.DBF)

2 (ODBC):
ODBC datasource (e.g. "ALIAS: MYDB1") or
ODBC DSN file (e.g. "C:\DB\mydb.dsn") or
ODBC connection string
(example: "DRIVER=SQL Server;SERVER=snake\sqlexpress;DATABASE=mydb")

3,4 (Firebird/Interbase):
connection string (example: "localhost:C:\DB\mydb.fdb")

role, charset:
Firebird/Interbase method only

memocp:
Codepage translation for MEMO fields (BDE, ODBC methods only)
0 - ANSI, 1-UTF8, 2-UNICODE_FSS (Firebird/Interbase), 3-Unicode, 4-Unicode reversed
It can also be any codepage number used by MultiByteToWideChar() Windows API function.

writemode:
1: read-write connection
0: read only connection

Returns:
dbnum > 1000 (success), dbnum = 1000 (error)

Important:
The database numbers from 0 to 999 are reserved for connections designed in the report project in REPCoder.

81) INT **DB_DISCONNECT**(INT dbnum)

Closes a database connection.
The dbnum must be an integer > 1000 returned by DB_CONNECT function.

Returns: 1 (success), 0 (error).

82) INT **SET_CONNECT_OPT**(INT dbnum, INT method, TEXT dbname, TEXT user, TEXT pass, TEXT role, TEXT charset, INT memocp, INT writemode)

Allows to change the connect options for a connection designed in the report form.
The function should be called at the same beginning, when database connections are still inactive.

The dbnum=1, 2 , 3, ... specifies the 1-based database number.
The arguments are described in details in the description of DB_CONNECT function.

If you set integer arguments: method, memocp, writemode as NULL they will remain unchanged.
If you set text arguments: dbname, user, pass, role, charset as 'NULL' they will remain unchanged.

Returns: 1 (success), 0 (error - bad dbnum argument).

83) INT **EXECSQL**(TEXT sql)

This function is for REPCoder only.
Executes SQL query in the default underlying database. If error occurs it is displayed in the message box window.

Returns: 1 (success), 0 (error).

Important:
The database connection must be opened in read-write mode.

84) INT **EXECSQL2**(INT dbnum, TEXT sql, INT show_error_message)

Executes SQL query in the given database. The error message box is displayed only if the show_error_message argument is nonzero.

REPCoder:
The dbnum argument specifies the 1-based database number designed in the report project. The dbnum=0 value corresponds to the default database of the underlying SQL query.

RCALC:
The dbnum can also be an integer > 1000 returned by DB_CONNECT function.

Returns: 1 (success), 0 (error).

Important:
The database connection must be opened in read-write mode.

85) INT **QUERY_LOAD**(INT dbnum, TEXT sql, INT show_error_message)

Executes SQL query of a "select" type in the given database. The error message box is displayed only if the show_error_message argument is nonzero.

REPCoder:
The dbnum argument specifies the 1-based database number as designed in the report form. The dbnum=0 value corresponds to the default database of the underlying SQL query.

RCALC:
The dbnum can also be an integer > 1000 returned by DB_CONNECT function.

The function loads the whole result set to memory and returns its handle as integer. Then the other functions can use this handle:
QUERY_FREE, QUERY_COLS, QUERY_ROWS, QUERY_COLNAME, QUERY_COLTYPE, QUERY_COLTYPENAME, QUERY_RESULT, QUERY_RESULT2.

Returns: handle to the result set (success), 0 (error).

Important:
The function supports only the queries starting with the "select" keyword.

86) INT **QUERY_FREE**(INT query_handle)

It frees the resources allocated by QUERY_LOAD.

Important:
The function releases resources immediately in the runtime. It is a good programming practice to do it. However it is not absolutely necessary here. If the function is not called, the program will free the memory at the end of the script execution.

87) INT **QUERY_COLS**(INT query_handle)

Returns the number of columns in the result set produced by the QUERY_LOAD function.

88) INT **QUERY_ROWS**(INT query_handle)

Returns the number of rows in the result set produced by the QUERY_LOAD function.

89) TEXT **QUERY_COLNAME**(INT query_handle, INT colnr)

Returns the name of the column in the result set produced by the QUERY_LOAD function. The column number is the input argument.

Important:
The column numbers start from 1 (1, 2, 3, ...).

90) INT **QUERY_COLTYPE**(INT query_handle, INT colnr)

Returns data type of the column in the result set produced by QUERY_LOAD function. The column number is the input argument.

Returns the following SQL type identifiers:
1 - char
2 - date
3 - time
4 - timestamp
5 - smallint
6 - int
7 - blob
8 - memo
9 - float, double, numeric

Important:
The column numbers start from 1 (1, 2, 3, ...). For unknown data types the function returns 0.

91) TEXT **QUERY_COLTYPENAME**(INT query_handle, INT colnr)

Returns the name of the data type of the column in the result set produced by the QUERY_LOAD function. The column number is the input argument.

Returns the following type names:
char, date, time, timestamp, smallint, int, blob, memo, float

Important:
The column numbers start from 1 (1, 2, 3, ...). For unknown data types the function returns empty text.

92) TEXT **QUERY_RESULT**(INT query_handle, INT rownum, INT colnr)

Returns the value of the result set produced by the QUERY_LOAD function. The row and column numbers are the input arguments.

Important:
The row and column numbers start from 1 (1, 2, 3, ...).
The function returns the data as text.
The format of date and time is: rrrr-mm-dd, hh:mm:ss.
For floating point numbers the dot character is the decimal separator.
To obtain original data types you can use the conversion functions: tonumber(), todate().

93) TEXT **QUERY_RESULT2**(INT query_handle, INT rownum, TEXT colname)

This function is very similar to the QUERY_RESULT function. The only difference is that it requires the column name as the input argument instead of the column number.

Returns the value of the result set produced by the QUERY_LOAD function. Row number and column name are the input arguments.

Important:
The row numbers start from 1 (1, 2, 3, ...).
The function returns the data as text.
The format of date and time is: rrrr-mm-dd, hh:mm:ss.
For floating point numbers the dot character is the decimal separator.
To obtain original data types you can use the conversion functions: tonumber(), todate().

94) TEXT **SELECT_SINGLE**(INT dbnum, TEXT sql, INT show_error_message)

Executes SQL query of a "select" type in the given database. The error message box is displayed only if the show_error_message argument is nonzero.

The dbnum argument specifies the 1-based database number as designed in the report form.
The dbnum=0 value corresponds to the default database of the underlying SQL query.
The dbnum can also be an integer > 1000 returned by DB_CONNECT function.

The function returns only the first row of the result set !!!

The row is returned as text. If the query produces more than one column, they are separated with the ### separator. To obtain a single column you should use the getfield() function. The single columns are returned as texts exactly as in the QUERY_RESULT function.
The format of date and time is: rrrr-mm-dd, hh:mm:ss.
For floating point numbers the dot character is the decimal separator.
To obtain original data types you can use the conversion functions: tonumber(), todate().

95) TEXT **getfield**(TEXT row, int fieldnr)

This function usually works together with the SELECT_SINGLE function, when it returns more than one column in the row.

Returns the column value identified by the fieldnr, in the row.

Important:
The single columns in the row are separated with: ###
The row numbers start from 1 (1, 2, 3, ...).
The function works the same as the GETPARAMFIELD function.

Example:
```
TEXT T = 'Adam###Smith###1970-12-20###1234.56';

TEXT firstname = getfield(T, 1);
TEXT lastname = getfield(T, 2);
TEXT birthdate = getfield(T, 3);
TEXT money = getfield(T, 4);
```

96) INT **getfieldcount**(TEXT row)

Returns the number of fields in the text row.
The fields are assumed to be separated with: ###

97) INT **TAB_ALLOC**(INT nitems, INT item_size)

Creates a memory table of nitems elements (items). The size (in bytes) of a single element is item_size. The initial value of nitems can be equal 0 (the table can be expanded later). For the table of numerical values it is recomended to use item_size = 8 (C language: double, __int64). You can also use item_size = 4 for INT type or even less (1, 2) if the integer values stored in the table are small.

Returns: unique table number = 1,2,3 ... (success), 0 (error)

Important:
- After the table is used it should be freed with the TAB_FREE function. It is recommended, but optional, because at the end of the script execution the program frees all active tables to avoid memory corruption.

- All the functions of memory tables are safe to use. They do not generate memory errors if the input arguments are bad.

- You can also store texts in a table. For this you should know, that the size of a single character is 2 bytes (UNICODE wchar_t type). So you should define item_size twice as big as the maximum number of characters you want to store. Otherwise texts are truncated. Do not care about the terminating (2-byte) null character. It is not stored with texts in memory tables.

98) INT **TAB_REALLOC**(INT tabnum, INT new_nitems)

Reallocates the table to new_nitems elements. The tabnum is a table identifier returned by TAB_ALLOC function.

Returns: 1 (success), 0 (error)

99) INT **TAB_FREE**(INT tabnum)

Frees the table. If you don't do it, the program will do it automatically at the end of script execution.

Returns: 1 (success), 0 (error)

100) NUMERIC **TAB_GETITEM**(INT tabnum, INT pos, INT numeric_scale)

Returns a numerical value (FLOAT or NUMERIC type) depending on the numeric_scale argument of a table item. The allowed values of the scale are:
-1: FLOAT
0,1,2, ... 18: NUMERIC

The argument pos is a zero-based position (like in the C language) of the table element.

Returns the value of the element on success or NULL value on error (for example: out-of-range pos argument or bad tabnum or numeric_scale)

101) TEXT **TAB_GETTEXTITEM**(INT tabnum, INT pos)

Retrieves the item value in the table of texts.
Returns a text value on success or empty text on bad argument or error.

102) INT **TAB_SETITEM**(INT tabnum, INT pos, ??? value)

Sets the value of the item at position pos in the table. The argument value can be a numeric type (INT, FLOAT, NUMERIC) or a TEXT type (in the case of text table).

Important:

The data from a value to table is copied in the original form, without any floating point conversion. So you must provide a correct numeric scale in the TAB_GETITEM function to retrieve the correct value.

Returns: 1 (success), 0 (error)

103) INT **TAB_SET**(INT tabnum, INT pos, INT nitems, ??? value)

Sets the nitems elements of a table starting at position (0-based) pos. The argument value can be a number or a text. It works like a memset C function.

Returns: 1 (success), 0 (error)

104) INT **TAB_APPEND**(INT tabnum, ??? value)

Append a single item at the end of a table. The argument value can be a number or a text.

Returns: 1 (success), 0 (error)

105) INT **TAB_INSERT**(INT tabnum, INT pos, ??? value)

Inserts a single item into a table at position (0-based) pos. The argument value can be a number or a text.

Returns: 1 (success), 0 (error)

106) INT **TAB_APPEND2**(INT tabnum1, INT tabnum2, INT pos2, INT nitems)

Appends nitems elements at the end of the destination table tabnum1. The items are read from the source table tabnum2, starting from the position pos2.

Returns: 1 (success), 0 (error)

107) INT **TAB_INSERT2**(INT tab1, INT pos1, INT tab2, INT pos2, INT nitems)

Inserts nitems elements at position pos1 of the destination table tab1. The items are read from the source table tab2, starting from the position pos2.

Returns: 1 (success), 0 (error)

108) INT **TAB_REMOVE**(INT tabnum, INT pos, INT nitems)

Removes nitems elements from a table at position (0-based) pos.

Returns: 1 (success), 0 (error)

109) INT **TAB_GETCOUNT**(INT tabnum)

Returns the number of elements (not bytes!!!) stored in the table.

110) INT **TAB_GET_ITEMSIZE**(INT tabnum)

Returns the size (in bytes) of a single element of the table.

111) INT **TAB_CHANGE_ITEMSIZE**(INT tabnum, INT new_item_size)

Allows to changes the size (in bytes) of a single element of the table. It can only be successfull if the table is empty or if the total size of bytes stored in the table is a multiplication of new_item_size.

Returns: 1 (success), 0 (error)

112) INT **TAB_COPY**(INT tab1, INT pos1, INT tab2, INT pos2, INT nitems)

Copies data (nitems elements) between 2 tables (like memcpy in C language).

Destination table: tab1, pos1
Source table: tab2, pos2

Important:
If nitems<0 then all items starting at pos2 from the source table are copied.

113) INT **TAB_READ**(INT tabnum, INT pos, INT nitems, TEXT fname, INT fpos)

Reads binary data (nitems elements) from a file to a table. The fpos argument (file position) is in the item_size table units (not in bytes!!!). If nitems<0 the entire file is read, assuming that its size is a multiplication of the item_size.

Important:
If the table is empty, it will be automatically allocated by the program.

Returns: the number of read items (success), 0 (error)

114) INT **TAB_WRITE**(INT tab, INT pos, INT nitems, TEXT fname, INT append)

Writes binary data (nitems elements) from a table to a file. If nitems<0 the entire table is written from the pos to the end.

Returns the number of written items (success), 0 (error)

115) INT **TAB_FROMFILE**(TEXT fname)

Creates a memory table of bytes read from a file. The size of a single element (item_size) is 1 byte. It works like TAB_ALLOC function.

Returns a unique tabnum = 1, 2, 3, ... (success), 0 (error)

116) INT **TAB_TOFILE**(INT tabnum, TEXT fname)

Writes all data from a memory table to a file.
Returns: 1 (success), 0 (error).

117) INT **PUT_BLOB**(INT dbnum, TEXT TableName, TEXT FieldName, TEXT WhereSql, INT tabnum, INT show_error_message)

Puts blob data (stored in memory table: tabnum) to a database table at the given field, for the record specified in the WhereSql clause. If tabnum=0 the blob field is set to NULL.

The dbnum argument specifies the 1-based database number as designed in the report form.
The dbnum=0 value corresponds to the default database of the underlying SQL query.
The dbnum can also be an integer > 1000 returned by DB_CONNECT function.

Returns: 1 (success), 0 (error).

118) INT **PUT_MEMO**(INT dbnum, TEXT TableName, TEXT FieldName, TEXT WhereSql, TEXT Memo, INT show_error_message)

Puts BLOB MEMO data (stored in the TEXT variable: Memo) to a database table at the given field, for the record specified in the WhereSql clause. If Memo = '' (empty text) the field is set to NULL.

The dbnum argument specifies the 1-based database number as designed in the report form.
The dbnum=0 value corresponds to the default database of the underlying SQL query.
The dbnum can also be an integer > 1000 returned by DB_CONNECT function.

Returns: 1 (success), 0 (error).

119) TEXT **currdir**()

Returns the current directory.

120) INT **random**(INT n)

Returns a random number between 0 and n-1.

121) INT **file_len**(TEXT fname)

Returns the size in bytes of a file.

122) INT **file_exists**(TEXT fname)

Checks if the file exists.
Returns: 1 (file exists), 0 (file does not exists).

123) INT **DELETE_FILE**(TEXT fname)

Deletes a file.
Returns: 1 (success), 0 (error).

124) TEXT **get_odbc_drivers**()

Returns the list of ODBC driver names in the form of a TEXT. Driver names are separated with: ###, like fields returned by SELECT_SINGLE. Use getfield(drv, i) function to get the i-th driver name.

Example:

```
TEXT drv = get_odbc_drivers();
INT n = getfieldcount(drv);
INT i = 1;
WHILE(i <= n)
  {
   WARNING('Driver name nr ' + totext(i) + ': ' + getfield(drv, i));
   i = i + 1;
  }
```

125) TEXT **find_files**(TEXT filter)

Returns a list of file names in the form of a TEXT. File names are separated with: ###, like fields returned by SELECT_SINGLE. Use getfield(files, i) function to get the i-th file name.

Example:

```
TEXT files = find_files('*.bmp');
INT n = getfieldcount(files);
INT i = 1;
WHILE(i <= n)
```

```
{
 WARNING('File name nr ' + totext(i) + ': ' + getfield(files, i));
 i = i + 1;
}
```

126) INT **convert_image**(INT src_tabnum, INT image_type, INT jpg_quality)

Copies a source memory table (src_tabnum) containing image data to a new memory table, making a conversion of image format. You can for example convert a BMP to JPEG. The function is using GDIPLUS Windows functions.

You specify the desired image format in the image_type parameter. It must be one of the following 5 values:
1 – BMP, 2 – JPG, 3 – GIF, 4 – TIFF, 5 - PNG

For the JPG format you must also specify the jpg_quality (from 0 to 100).

Returns a unique table number = 1, 2, 3, ... (success), 0 (error)

19. Exported functions of the RCALC API

As it was already mentioned at the same beginning the RCALC module is a part of **REPCoder.DLL**. It is a subset of exported functions in this reporting library. The RCALC API functions are completely independent on the other exported functions - the main reporting API. So they can be used by programmers of any applications, that have nothing to do with reporting or databases. The API is thread-safe. The number of functions is 86, but most of them have both versions: UNICODE and ANSI. The ANSI version has the name ending with **_a** just like for all other exported functions of REPCoder.DLL. The functions are using the WINAPI calling convention like standard WIN32 API functions. The header file name is **repcoder.h**. To build an application it is also required to include **repcoder.c** in the project.

All the function names of the RCALC API begin with **rcalc_**. They are fully independent on other REPCoder exported functions. There are however 5 important functions that are common to REPCoder and RCALC:

- **repcoder_load**

Loads REPCoder.DLL (calls LoadLibrary).

- **repc_init**

Initializes REPCoder or RCALC.

- **repc_free_memory**

Frees memory allocated by **rcalc_alloc_error_text.**

- **repc_exit**

Clears all allocated resources.

- **repcoder_free**

Frees REPCoder.DLL (calls FreeLibrary).

So the application template looks like:

```
repcoder_load();  // only in C and C++
repc_init();

Call RCALC or REPCoder API functions

repc_exit();
repcoder_free();  // only in C and C++
```

The same application template is also used for other programming languages (C#, Java, Delphi) but without the functions repcoder_load, repcoder_free. These languages load and free REPCoder.dll automatically. It is shown in the examples in next chapters.

RCALC API function reference:

1) int WINAPI **rcalc_alloc_handle**(void)

Allocates the RCALC handle creating a resource in memory. This handle is used later by other API functions to compile, execute, get results and free the allocated resource. The thread that called this function becomes the owner thread of the handle. Only the owner thread can call the other API functions for this handle. There is only one exception - another thread can call the **rcalc_copy_handle** function to become the owner of the copied handle.

Returns a unique integer handle value > 0 on success or 0 on error.

2) int WINAPI **rcalc_copy_handle**(int calc_handle)

Creates a copy of the existing handle making a duplicated resource. It is useful when you want to use a copy of the handle in another thread. The thread that called this function becomes the owner thread of the handle.

Returns a unique integer handle value > 0 on success or 0 on error.

3) BOOL WINAPI **rcalc_free_handle**(int calc_handle)

Frees the handle returned by **rcalc_alloc_handle** or **rcalc_copy_handle**. Only the owner thread of the handle can call this function with success.

Returns 1 (success) or 0 (error).

4) BOOL WINAPI **rcalc_free_all_handles**(void)

Frees all the handles allocated by the module. It is not necessary to call this function if **rcalc_free_handle** was called for each allocated handle. The resources are also automatically released when the REPCoder module exits (calling the **repc_exit** function).

5) int WINAPI **rcalc_get_state**(int calc_handle)

Returns the state of the handle:
1 – allocated, 2 – compiled, 3 – executed, 0 – freed or invalid handle

6) BOOL WINAPI **rcalc_is_valid**(int calc_handle)

Returns: 1 (handle is valid) or 0 (handle is invalid).

7) BOOL WINAPI **rcalc_is_compiled**(int calc_handle)

Returns: 1 if the handle is valid and was already compiled or 0 otherwise.

8) BOOL WINAPI **rcalc_is_executed**(int calc_handle)

Returns: 1 if the handle is valid and was already executed or 0 otherwise.

9) void WINAPI **rcalc_set_status_window**(int calc_handle, HWND hStatWnd)

Sets the status window for the handle. It is used by the internal **STATUSINFO** function of the RCALC language. The default status window is the program Console, but it can be set to any other window handle. The status window is useful when you need to display some progress information in the runtime when the algorithm is executed.

10) BOOL WINAPI **rcalc_compile**(int calc_handle, const wchar_t* pText)
11) BOOL WINAPI **rcalc_compile_a**(int calc_handle, const char* pText)

One of the most important functions of the module. It compiles the text of the algorithm.
Returns: 1 (success) or 0 (error).

12) BOOL WINAPI **rcalc_compile_file**(int calc_handle, const wchar_t* FileName, int Format)
13) BOOL WINAPI **rcalc_compile_file_a**(int calc_handle, const char* FileName, int Format)

Compiles an algorithm that is stored in a text file.

The format parameter can have the following values:
0- ANSI, 1- UTF8, 2- UTF8 (+BOM), 3- Unicode, 4- Unicode reversed.
Its value can also be any codepage number used by WideCharToMultiByte() Windows API function.

Returns: 1 (success) or 0 (error).

14) void WINAPI **rcalc_show_error**(HWND hParent)

Displays the last RCALC error message for the current thread in a MessageBox. You can set a parent window of this MessageBox in the hParent argument.

15) int WINAPI **rcalc_get_error_text_len**(void)

Returns the length (number of characters) of the last RCALC error message for the current thread. This information can be used to allocate the application buffer for the message. The buffer size should also include the terminating null character:

```
int bufsize = (1+length)*sizeof(wchar_t);        // UNICODE version
int bufsize = (1+length)*sizeof(char);           // ANSI version
```

If no error message is available for the current thread the function returns 0.

16) BOOL WINAPI **rcalc_copy_error_text**(wchar_t* buf, int len)
17) BOOL WINAPI **rcalc_copy_error_text_a**(char* buf, int len)

Copies the last RCALC error message text for the current thread to the application buffer. The second argument (len) is the size of the buffer.

Returns: 1 (success) or 0 (error).

18) const wchar_t* **rcalc_alloc_error_text**(void)
19) const char* **rcalc_alloc_error_text_a**(void)

Allocates a buffer and fills it with the last RCALC error message text for the current thread. Returns the non-zero address of the buffer. If there is no error message available the returned allocated address contains an empty string (a single NULL character). So the function returns 0 only in the case of internal error.

Important:
The application should free the returned value calling the **repc_free_memory** function.

20) int WINAPI **rcalc_get_compiled_text_len**(int calc_handle, BOOL bCRLF)

Returns the length (number of characters) of the compiled text for the handle. The second argument (bCRLF) specifies the desired form of new line characters. If the handle was not compiled the function returns 0.
This information can be used to allocate the application buffer for the compiled text. The buffer size should also include the terminating null character:

```
int bufsize = (1+length)*sizeof(wchar_t);        // UNICODE version
int bufsize = (1+length)*sizeof(char);           // ANSI version
```

21) BOOL WINAPI **rcalc_copy_compiled_text**(int calc_handle, BOOL bCRLF, wchar_t* buf, int len)
22) BOOL WINAPI **rcalc_copy_compiled_text_a**(int calc_handle, BOOL bCRLF, char* buf, int len)

Copies the compiled text for the handle to the application buffer. The second argument (bCRLF) specifies the desired form of new line characters. The last argument (len) is the size of the buffer. It is often useful to display the compiled text of the script in a window to see how RCALC formatted the source text.

Returns: 1 (success) or 0 (error).

23) const wchar_t* WINAPI **rcalc_get_compiled_text**(int calc_handle, BOOL bCRLF)
24) const char* WINAPI **rcalc_get_compiled_text_a**(int calc_handle, BOOL bCRLF)

Returns the RCALC internal pointer to the compiled text for the handle. The second argument (bCRLF) specifies the desired form of new line characters.
If the handle was not compiled the returned pointer points to a const empty string (a single NULL character). So the function returns 0 only in the case of internal error.

Important:
The returned const pointer points to internal RCALC resources. It cannot be written or freed by the application !!!

25) BOOL WINAPI **rcalc_execute**(int calc_handle, int nInput, const wchar_t* const* pInputValues)
26) BOOL WINAPI **rcalc_execute_a**(int calc_handle, int nInput, const char* const* pInputValues)

One of the most important functions of the module. It executes the compiled handle passing optional input values for the algorithm.

The values passed here from the calling application are in the form of a table of C-string pointers.

Returns: 1 (success) or 0 (error).

27) BOOL WINAPI **rcalc_execute2**(int calc_handle, const wchar_t* InputValues, const wchar_t* Separator)
28) BOOL WINAPI **rcalc_execute2_a**(int calc_handle, const char* InputValues, const char* Separator)

One of the most important functions of the module. It executes the compiled handle passing optional input values for the algorithm.

The values passed here from the calling application are in the form of a C-string. They are separated with the last argument (Separator) inside the host string (InputValues). If the Separator is NULL or points to an empty string, the standard 3-hash separator ### will be assumed.

Returns: 1 (success) or 0 (error).

29) int WINAPI **rcalc_get_input_count**(int calc_handle)

Returns the number of input variables for the compiled handle. It is just the number of variables defined in the INPUT section of the algorithm. If the handle was not compiled the function returns 0.

30) int WINAPI **rcalc_get_input_name_len**(int calc_handle, int number)

Returns the length (number of characters) of the name of the specified input variable of the algorithm (number is the 1-based ordinal position in the INPUT section). If the handle was not compiled the function returns 0.
This information can be used to allocate the application buffer for the input name. The buffer size should also include the terminating null character:

```
int bufsize = (1+length)*sizeof(wchar_t);        // UNICODE version
int bufsize = (1+length)*sizeof(char);           // ANSI version
```

31) BOOL WINAPI **rcalc_copy_input_name**(int calc_handle, int number, wchar_t* buf, int len)
32) BOOL WINAPI **rcalc_copy_input_name_a**(int calc_handle, int number, char* buf, int len)

Copies the name of the specified input variable of the algorithm (number is the 1-based ordinal position in the INPUT section) to the application buffer. The last argument (len) is the size of the buffer.

Returns: 1 (success) or 0 (error).

33) const wchar_t* WINAPI **rcalc_get_input_name**(int calc_handle, int number)
34) const char* WINAPI **rcalc_get_input_name_a**(int calc_handle, int number)

Returns the RCALC internal pointer to the name of the specified input variable of the algorithm (number is the 1-based ordinal position in the INPUT section).
If the handle was not compiled the returned pointer points to a const empty string (a single NULL character). So the function returns 0 only in the case of internal error.

Important:
The returned const pointer points to internal RCALC resources. It cannot be written or freed by the application !!!

35) int WINAPI **rcalc_get_input_number**(int calc_handle, wchar_t* name)
36) int WINAPI **rcalc_get_input_number_a**(int calc_handle, char* name)

Returns the 1-based position in the INPUT section of the specified input variable (name is the name of the input variable).

37) int WINAPI **rcalc_get_input_type**(int calc_handle, int number)

Returns the type of the specified input variable of the algorithm (number is the 1-based ordinal position in the INPUT section) for the compiled handle:

1 – TEXT

2 – FLOAT
3 – INT
4 – DATE
5 – TIME
6 – TIMESTAMP
10 – NUMERIC(0)
11 – NUMERIC(1)
…
18 – NUMERIC(18)

If the handle is not compiled or invalid input number is specified the function returns 0.

38) int WINAPI **rcalc_get_input_type_numscale**(int calc_handle, int number)

Returns the numeric scale of the specified input variable of the algorithm (number is the 1-based ordinal position in the INPUT section) for the compiled handle. The returned scale value can be: 0, 1, 2, …, 18.

If the handle is not compiled or invalid input number is specified the function returns 0.

39) int WINAPI **rcalc_get_input_type_name_len**(int calc_handle, int number)

Returns the length (number of characters) of the type name of the specified input variable of the algorithm (number is the 1-based ordinal position in the INPUT section). If the handle was not compiled the function returns 0.
This information can be used to allocate the application buffer for the input type name. The buffer size should also include the terminating null character:

```
int bufsize = (1+length)*sizeof(wchar_t);        // UNICODE version
int bufsize = (1+length)*sizeof(char);           // ANSI version
```

40) BOOL WINAPI **rcalc_copy_input_type_name**(int calc_handle, int number, wchar_t* buf, int len)
41) BOOL WINAPI **rcalc_copy_input_type_name_a**(int calc_handle, int number, char* buf, int len)

Copies the type name of the specified input variable of the algorithm (number is the 1-based ordinal position in the INPUT section) to the application buffer. The last argument (len) is the size of the buffer.

Returns: 1 (success) or 0 (error).

42) const wchar_t* WINAPI **rcalc_get_input_type_name**(int calc_handle, int number)
43) const char* WINAPI **rcalc_get_input_type_name_a**(int calc_handle, int number)

Returns the RCALC internal pointer to the type name of the specified input variable of the algorithm (number is the 1-based ordinal position in the INPUT section).
If the handle was not compiled the returned pointer points to a const empty string (a single NULL character). So the function returns 0 only in the case of internal error.

Important:
The returned const pointer points to internal RCALC resources. It cannot be written or freed by the application !!!

44) int WINAPI **rcalc_get_output_count**(int calc_handle)

Returns the number of output variables for the compiled handle. It is just the number of variables defined in the OUTPUT section of the algorithm. If the handle was not compiled the function returns 0.

45) int WINAPI **rcalc_get_output_name_len**(int calc_handle, int number)

Returns the length (number of characters) of the name of the specified output variable of the algorithm (number is the 1-based ordinal position in the OUTPUT section). If the handle was not compiled the function returns 0.
This information can be used to allocate the application buffer for the output name. The buffer size should also include the terminating null character:

```
int bufsize = (1+length)*sizeof(wchar_t);        // UNICODE version
int bufsize = (1+length)*sizeof(char);           // ANSI version
```

46) BOOL WINAPI **rcalc_copy_output_name**(int calc_handle, int number, wchar_t* buf, int len)
47) BOOL WINAPI **rcalc_copy_output_name_a**(int calc_handle, int number, char* buf, int len)

Copies the name of the specified output variable of the algorithm (number is the 1-based ordinal position in the OUTPUT section) to the application buffer. The last argument (len) is the size of the buffer.

Returns: 1 (success) or 0 (error).

48) const wchar_t* WINAPI **rcalc_get_output_name**(int calc_handle, int number)
49) const char* WINAPI **rcalc_get_output_name_a**(int calc_handle, int number)

Returns the RCALC internal pointer to the name of the specified output variable of the algorithm (number is the 1-based ordinal position in the OUTPUT section).
If the handle was not compiled the returned pointer points to a const empty string (a single NULL character). So the function returns 0 only in the case of internal error.

Important:
The returned const pointer points to internal RCALC resources. It cannot be written or freed by the application !!!

50) int WINAPI **rcalc_get_output_number**(int calc_handle, const wchar_t* name)
51) int WINAPI **rcalc_get_output_number_a**(int calc_handle, const char* name)

Returns the 1-based position in the OUTPUT section of the specified input variable (name is the name of the output variable).

52) int WINAPI **rcalc_get_output_type**(int calc_handle, int number)

Returns the type of the specified output variable of the algorithm (number is the 1-based ordinal position in the OUTPUT section) for the compiled handle:

1 – TEXT
2 – FLOAT
3 – INT
4 – DATE
5 – TIME
6 – TIMESTAMP
10 – NUMERIC(0)
11 – NUMERIC(1)
…
18 – NUMERIC(18)

If the handle is not compiled or invalid output number is specified the function returns 0.

53) int WINAPI **rcalc_get_output_type_textlen**(int calc_handle, int number)

Returns the text length of the specified output variable of the algorithm (number is the 1-based ordinal position in the OUTPUT section) for the compiled handle.

If the output TEXT variable has not specified the text length parameter (its length is unlimited) the function returns 0. If the handle is not compiled or invalid output number is specified the function returns 0.

54) int WINAPI **rcalc_get_output_type_numscale**(int calc_handle, int number)

Returns the numeric scale of the specified output variable of the algorithm (number is the 1-based ordinal position in the OUTPUT section) for the compiled handle. The returned scale value can be: 0, 1, 2, …, 18.

If the handle is not compiled or invalid output number is specified the function returns 0.

55) int WINAPI **rcalc_get_output_type_name_len**(int calc_handle, int number)

Returns the length (number of characters) of the type name of the specified output variable of the algorithm (number is the 1-based ordinal position in the OUTPUT section). If the handle was not compiled the function returns 0.
This information can be used to allocate the application buffer for the output type name. The buffer size should also include the terminating null character:

```
int bufsize = (1+length)*sizeof(wchar_t);       // UNICODE version
int bufsize = (1+length)*sizeof(char);          // ANSI version
```

56) BOOL WINAPI **rcalc_copy_output_type_name**(int calc_handle, int number, wchar_t* buf, int len)

57) BOOL WINAPI **rcalc_copy_output_type_name_a**(int calc_handle, int number, char* buf, int len)

Copies the type name of the specified output variable of the algorithm (number is the 1-based ordinal position in the OUTPUT section) to the application buffer. The last argument (len) is the size of the buffer.

Returns: 1 (success) or 0 (error).

58) const wchar_t* WINAPI **rcalc_get_output_type_name**(int calc_handle, int number)
59) const char* WINAPI **rcalc_get_output_type_name_a**(int calc_handle, int number)

Returns the RCALC internal pointer to the type name of the specified output variable of the algorithm (number is the 1-based ordinal position in the OUTPUT section).
If the handle was not compiled the returned pointer points to a const empty string (a single NULL character). So the function returns 0 only in the case of internal error.

Important:
The returned const pointer points to internal RCALC resources. It cannot be written or freed by the application !!!

60) BOOL WINAPI **rcalc_isnull_input**(int calc_handle, int number)

Returns 1 if the specified input variable of the algorithm (number is the 1-based ordinal position in the INPUT section) is NULL or 0 if it is not NULL.

61) const wchar_t* WINAPI **rcalc_get_input_string**(int calc_handle, int number)
62) const char* WINAPI **rcalc_get_input_string_a**(int calc_handle, int number)

Returns the RCALC internal pointer to the value of the specified input variable of the algorithm (number is the 1-based ordinal position in the INPUT section). This value is passed from the calling application when the handle is executed.
If the handle was not executed the returned pointer points to a const empty string (a single NULL character). So the function returns 0 only in the case of internal error.

Important:
The returned const pointer points to internal RCALC resources. It cannot be written or freed by the application !!!

63) int WINAPI **rcalc_get_input_string_len**(int calc_handle, int number)

Returns the length (number of characters) of the value of the specified input variable of the algorithm (number is the 1-based ordinal position in the INPUT section). This value is passed from the calling application when the handle is executed.
If the handle was not executed the function returns 0.
This information can be used to allocate the application buffer for the input value. The buffer size should also include the terminating null character:

```
int bufsize = (1+length)*sizeof(wchar_t);        // UNICODE version
int bufsize = (1+length)*sizeof(char);           // ANSI version
```

64) int WINAPI **rcalc_copy_input_string**(int calc_handle, int number, wchar_t* buf, int len)
65) int WINAPI **rcalc_copy_input_string_a**(int calc_handle, int number, char* buf, int len)

Copies the value of the specified input variable of the algorithm (number is the 1-based ordinal position in the INPUT section) to the application buffer. This value is passed from the calling application when the handle is executed. The last argument (len) is the size of the buffer.
If the handle was not executed the function returns 0.

Returns: 1 (success) or 0 (error).

66) double WINAPI **rcalc_get_input_double**(int calc_handle, int number)

Returns the value of the specified input variable of the algorithm (number is the 1-based ordinal position in the INPUT section) converted to double. This value is passed from the calling application when the handle is executed. If the value cannot be converted to double the function returns 0. It works only for input variables of types: FLOAT, INT, NUMERIC.

67) int WINAPI **rcalc_get_input_int**(int calc_handle, int number)

Returns the value of the specified input variable of the algorithm (number is the 1-based ordinal position in the INPUT section) converted to int. This value is passed from the calling application when the handle is executed. If the value cannot be converted to int the function returns 0. It works only for input variables of types: FLOAT, INT, NUMERIC.

68) __int64 WINAPI **rcalc_get_input_numeric**(int calc_handle, int number)

Returns the value of the specified input variable of the algorithm (number is the 1-based ordinal position in the INPUT section) converted to __int64. This value is passed from the calling application when the handle is executed. If the value cannot be converted to __int64 the function returns 0. It works only for input variables of types: FLOAT, INT, NUMERIC.

69) __int64 WINAPI **rcalc_get_input_time**(int calc_handle, int number)

Returns the value of the specified input variable of the algorithm (number is the 1-based ordinal position in the INPUT section) converted to __int64. This value is passed from the calling application when the handle is executed. It works only for input variables of types: DATE, TIME, TIMESTAMP. Otherwise the function returns 0. The 64-bit value returned by this function encodes date, time or timestamp. It can be decoded with functions:

rcalc_get_year, rcalc_get_month, rcalc_get_day, rcalc_get_hour, rcalc_get_minute, rcalc_get_second, rcalc_get_millisec

70) BOOL WINAPI **rcalc_isnull_output**(int calc_handle, int number)

Returns 1 if the specified output variable of the algorithm (number is the 1-based ordinal position in the OUTPUT section) is NULL or 0 if it is not NULL.

71) const wchar_t* WINAPI **rcalc_get_output_string**(int calc_handle, int number)
72) const char* WINAPI **rcalc_get_output_string_a**(int calc_handle, int number)

Returns the RCALC internal pointer to the value of the specified output variable of the algorithm (number is the 1-based ordinal position in the OUTPUT section). This value is just the result of calculations (converted to string) after the handle is executed.
If the handle was not executed the returned pointer points to a const empty string (a single NULL character). So the function returns 0 only in the case of internal error.

Important:
The returned const pointer points to internal RCALC resources. It cannot be written or freed by the application !!!

73) int WINAPI **rcalc_get_output_string_len**(int calc_handle, int number)

Returns the length (number of characters) of the value of the specified output variable of the algorithm (number is the 1-based ordinal position in the OUTPUT section). This value is just the result of calculations (converted to string) after the handle is executed.
If the handle was not executed the function returns 0.
This information can be used to allocate the application buffer for the output value. The buffer size should also include the terminating null character:

```
int bufsize = (1+length)*sizeof(wchar_t);        // UNICODE version
int bufsize = (1+length)*sizeof(char);           // ANSI version
```

74) int WINAPI **rcalc_copy_output_string**(int calc_handle, int number, wchar_t* buf, int len)
75) int WINAPI **rcalc_copy_output_string_a**(int calc_handle, int number, char* buf, int len)

Copies the value of the specified output variable of the algorithm (number is the 1-based ordinal position in the OUTPUT section) to the application buffer. This value is just the result of calculations (converted to string) after the handle is executed. The last argument (len) is the size of the buffer.
If the handle was not executed the function returns 0.

Returns: 1 (success) or 0 (error).

76) double WINAPI **rcalc_get_output_double**(int calc_handle, int number)

Returns the value of the specified output variable of the algorithm (number is the 1-based ordinal position in the OUTPUT section) converted to double. This value is just the result of calculations after the handle is executed. If the value cannot be converted to double the function returns 0. It works only for output variables of types: FLOAT, INT, NUMERIC.

77) int WINAPI **rcalc_get_output_int**(int calc_handle, int number)

Returns the value of the specified output variable of the algorithm (number is the 1-based ordinal position in the OUTPUT section) converted to int. This value is just the result of calculations after the handle is executed. If the value cannot be converted to int the function returns 0. It works only for output variables of types: FLOAT, INT, NUMERIC.

78) __int64 WINAPI **rcalc_get_output_numeric**(int calc_handle, int number)

Returns the value of the specified output variable of the algorithm (number is the 1-based ordinal position in the OUTPUT section) converted to __int64. This value is just the result of calculations after the handle is executed. If the value cannot be converted to __int64 the function returns 0. It works only for output variables of types: FLOAT, INT, NUMERIC.

79) __int64 WINAPI **rcalc_get_output_time**(int calc_handle, int number)

Returns the value of the specified output variable of the algorithm (number is the 1-based ordinal position in the OUTPUT section) converted to __int64. This value is just the result of calculations after the handle is executed. It works only for output variables of types: DATE, TIME, TIMESTAMP. Otherwise the function returns 0. The 64-bit value returned by this function encodes date, time or timestamp. It can be decoded with functions:

rcalc_get_year, rcalc_get_month, rcalc_get_day, rcalc_get_hour, rcalc_get_minute, rcalc_get_second, rcalc_get_millisec

80) int WINAPI **rcalc_get_year**(__int64 value)

Returns the **year** part of the DATE encoded in the 64-bit value returned by functions: **rcalc_get_input_time, rcalc_get_output_time**.

81) int WINAPI **rcalc_get_month**(__int64 value)

Returns the **month** part of the DATE encoded in the 64-bit value returned by functions: **rcalc_get_input_time, rcalc_get_output_time**.

82) int WINAPI **rcalc_get_day**(__int64 value)

Returns the **day** part of the DATE encoded in the 64-bit value returned by functions: **rcalc_get_input_time, rcalc_get_output_time**.

83) int WINAPI **rcalc_get_hour**(__int64 value)

Returns the **hour** part of the TIME encoded in the 64-bit value returned by functions: **rcalc_get_input_time, rcalc_get_output_time**.

84) int WINAPI **rcalc_get_minute**(__int64 value)

Returns the **minute** part of the TIME encoded in the 64-bit value returned by functions: **rcalc_get_input_time, rcalc_get_output_time**.

85) int WINAPI **rcalc_get_second**(__int64 value)

Returns the **second** part of the TIME encoded in the 64-bit value returned by functions: **rcalc_get_input_time, rcalc_get_output_time**.

86) int WINAPI **rcalc_get_millisec**(__int64 value)

Returns the **millisecond** part of the TIME encoded in the 64-bit value returned by functions: **rcalc_get_input_time, rcalc_get_output_time**.

20. A sample application in C

To write a simple RCALC application in C is very easy. It is shown in the following example.

```
/*
compile VC++: cl test.c repcoder.c user32.lib
compile BCC32: bcc32 test.c repcoder.c
*/

#include "repcoder.h"
#include <stdio.h>
//------------------------

// Text of an algorithm
const char* AlgTxt =
"INPUT {"
"FLOAT x;"
"FLOAT y;"
"}"
"FLOAT z = x + y;"
"OUTPUT {"
"FLOAT Out1 = z*z;"
"FLOAT Out2 = sqrt(x + y);"
"}";
//------------------------

int main(void)
{
int h, nOut;
BOOL ok;
double Out1, Out2;

// Load repcoder.dll
if( !repcoder_load() ) return 0;

// Init repcoder
if( !repc_init() ) { repcoder_free(); return 0; }

// Alloc RCALC handle
h = rcalc_alloc_handle();
if( !h ) { repc_exit(); repcoder_free(); return 0; }

// Compile RCALC handle (ANSI version: _a)
ok = rcalc_compile_a(h, AlgTxt);
if( !ok )
  {
   rcalc_show_error(0); // Show error in a MessageBox
   rcalc_free_handle(h); repc_exit(); repcoder_free();
```

```
   return 0;
  }

// Show compiled text in the Console
printf("\n******* Compiled text **************\n\n");
printf(rcalc_get_compiled_text_a(h, 1));
printf("\n\n******* End of compiled text ********\n\n");

// Get and print the number of output values
nOut = rcalc_get_output_count(h);
printf("nOut = %d\n", nOut);

// Execute with some input values: x = 3.14159, y = 2.71
ok = rcalc_execute2_a(h, "3.14159###2.71", 0);
if( !ok )
  {
   rcalc_show_error(0); rcalc_free_handle(h); repc_exit(); repcoder_free();
   return 0;
  }

// Print the results (UNICODE version)
wprintf(L"Out1 = %s\n", rcalc_get_output_string(h, 1));
wprintf(L"Out2 = %s\n\n", rcalc_get_output_string(h, 2));

// Get the results to host variables
Out1 = rcalc_get_output_double(h, 1);
Out2 = rcalc_get_output_double(h, 2);
printf("Out1 = %lf, Out2 = %lf\n", Out1, Out2);

// Free RCALC handle (not necessary, repc_exit() can also free it)
rcalc_free_handle(h);

// Exit repcoder
repc_exit();

// Free repcoder.dll
repcoder_free();
return 0;
}
```

To run the above sample **repcoder.dll** file must be deployed together with the EXE file.

21. A sample application in C++

For C++ programmers there is a simple class RCALC defined in the files **rcalc_class.h**, **rcalc_class.cpp**:

```
class RCALC
{
int Handle;

public:
RCALC();
RCALC(int OtherHandle);
RCALC(RCALC& rc);
~RCALC();
bool CopyFrom(RCALC& rc);
```

```
int  GetHandle() { return Handle; }
int  GetState();
bool IsValid();
bool IsCompiled();
bool IsExecuted();

void SetStatusWindow(HWND hStatWnd);

bool Compile(const wchar_t* pText);
bool Compile(const char* pText);
bool CompileFile(const wchar_t* FileName, int Format=0);
bool CompileFile(const char* FileName, int Format=0);

std::wstring CompiledText(bool bCRLF=1);
std::string CompiledTextA(bool bCRLF=1);

int InputCount();
int InputType(int number);
int InputTypeNumericScale(int number);
std::wstring InputTypeName(int number);
std::string InputTypeNameA(int number);
std::wstring InputName(int number);
std::string InputNameA(int number);

int OutCount();
int OutType(int number);
int OutTypeTextLen(int number);
int OutTypeNumericScale(int number);
std::wstring OutTypeName(int number);
std::string OutTypeNameA(int number);
std::wstring OutName(int number);
std::string OutNameA(int number);

bool Execute(int nInput, const wchar_t* const* pInputValues);
bool Execute(int nInput, const char* const* pInputValues);
bool Execute(const wchar_t* InputValues, const wchar_t* Separator=0);
bool Execute(const char* InputValues, const char* Separator=0);
bool Execute(int nInput, const std::wstring* pInputValues);
bool Execute(int nInput, const std::string* pInputValues);

bool InputIsNull(int number);
std::wstring InputString(int number);
std::string InputStringA(int number);
double InputDouble(int number);
int InputInt(int number);
__int64 InputNumeric(int number);
__int64 InputTime(int number);

bool OutIsNull(int number);
std::wstring OutString(int number);
std::string OutStringA(int number);
double OutDouble(int number);
int OutInt(int number);
__int64 OutNumeric(int number);
__int64 OutTime(int number);

static int Year(__int64 value) { return rcalc_get_year(value); }
static int Month(__int64 value) { return rcalc_get_month(value); }
static int Day(__int64 value) { return rcalc_get_day(value); }
static int Hour(__int64 value) { return rcalc_get_hour(value); }
```

```
static int Minute(__int64 value) { return rcalc_get_minute(value); }
static int Second(__int64 value) { return rcalc_get_second(value); }
static int Millisec(__int64 value) { return rcalc_get_millisec(value); }

static bool LastError();
static void ShowError(HWND hParent=0);
static std::wstring ErrorString();
static std::string ErrorStringA();
};
```

The RCALC class is using standard C++ classes **std::string** and **std::wstring**. There are both types of methods: UNICODE and ANSI. Their names are mostly the same (C++ overloading). Only a few methods have specific ANSI names (ended with A), because they return **std::string** type instead of **std::wstring**.

```
CompiledStringA, InputTypeNameA, InputNameA, OutTypeNameA, OutNameA,
InputStringA, OutStringA, ErrorStringA (static method)
```

A simple C++ example:

```
/*
compile VC++: cl /EHsc test.cpp rcalc_class.cpp repcoder.c user32.lib
compile BCC32: bcc32 rcalc_class.cpp test.cpp repcoder.c
*/

#include "rcalc_class.h"
#include <iostream>
//------------------------

// Text of an algorithm
const char* AlgTxt =
"INPUT {"
"FLOAT x;"
"FLOAT y;"
"}"
"FLOAT z = x + y;"
"OUTPUT {"
"FLOAT Out1 = z*z;"
"FLOAT Out2 = sqrt(x + y);"
"}";
//------------------------

int main(void)
{
// Load repcoder.dll
if( !repcoder_load() ) return 0;

// Init repcoder
if( !repc_init() ) { repcoder_free(); return 0; }

// Create RCALC object
RCALC rc;

// Compile RCALC object
bool ok = rc.Compile(AlgTxt);
if( !ok )
  {
   RCALC::ShowError(); // Show error in a MessageBox
   repc_exit();
```

```
   repcoder_free();
   return 0;
  }

// Show compiled text in the Console
std::cout<< "\n******* Compiled text ***************\n\n";
std::cout<< rc.CompiledTextA().c_str();
std::cout<< "\n\n******* End of compiled text ********\n\n";

// Get and print the number of output values
int nOut = rc.OutCount();
std::cout<< "nOut = " << nOut << "\n";

// Execute with some input values: x = 3.14159, y = 2.71
ok = rc.Execute("3.14159###2.71");
if( !ok )
  {
   RCALC::ShowError();
   repc_exit(); repcoder_free();
   return 0;
  }

// Print the results (UNICODE version)
std::wcout << L"Out1 = " << rc.OutString(1) << L"\n";
std::wcout << L"Out2 = " << rc.OutString(2) << L"\n\n";

// Get the results to host variables
double Out1 = rc.OutDouble(1);
double Out2 = rc.OutDouble(2);
std::cout << "Out1 = " << Out1 << ", Out2 = " << Out2 << "\n";

// Exit repcoder
repc_exit();

// Free repcoder.dll
repcoder_free();
return 0;
}
```

To run the above sample **repcoder.dll** file must be deployed together with the EXE file.

22. A sample application in C#

For C# programmers a simple class RCALC was defined in the **repcoder.cs** file. It allows to call the exported functions of **repcoder.dll**. The C# class is very similar to the C++ class.

```
class RCALC
{
int Handle;

public RCALC();
public RCALC(RCALC rc);
~RCALC();
public bool CopyFrom(RCALC rc);

public int  GetHandle();
public int  GetState();
```

```
public bool IsValid();
public bool IsCompiled();
public bool IsExecuted();

public void SetStatusWindow(IntPtr hStatWnd);

public bool Compile(string Source);
public bool CompileFile(string FileName, int Format);

public bool Execute(string[] InputValues);
public bool Execute(string InputValues, string Separator);
public bool Execute(string InputValues);
public bool Execute();

public string CompiledText(bool bCRLF);

public int InputCount();
public int InputType(int number);
public int InputTypeNumericScale(int number);
public string InputTypeName(int number);
public string InputName(int number);
public bool InputIsNull(int number);
public string InputString(int number);
public double InputDouble(int number);
public int InputInt(int number);
public long InputNumeric(int number);
public long InputTime(int number);

public int OutputCount();
public int OutType(int number);
public int OutTypeTextLen(int number);
public int OutTypeNumericScale(int number);
public string OutTypeName(int number);
public string OutName(int number);
public bool OutIsNull(int number);
public string OutString(int number);
public double OutDouble(int number);
public int OutInt(int number);
public long OutNumeric(int number);
public long OutTime(int number);

public static int Year(long value);
public static int Month(long value);
public static int Day(long value);
public static int Hour(long value);
public static int Minute(long value);
public static int Second(long value);
public static int Millisec(long value);

public static bool LastError();
public static void ShowError(IntPtr hParent);
public static string ErrorString();
}
```

A simple C# example:

```
//compile: csc test.cs repcoder.cs

using System;

class App
```

```
{
 public static void Main()
   {
// Init repcoder
    if( !Repcoder.init() )
      {
       Console.WriteLine("Repcoder.init() error ...");
       return;
      }

// Create RCALC object
    RCALC rc = new RCALC();

// Text of an algorithm
    string AlgTxt =
    "INPUT {" +
    "FLOAT x;" +
    "FLOAT y;" +
    "}" +
    "FLOAT z = x + y;" +
    "OUTPUT {" +
    "FLOAT Out1 = z*z;" +
    "FLOAT Out2 = sqrt(x + y);" +
    "}";

// Compile RCALC object
    bool ok = rc.Compile(AlgTxt);
    if( !ok )
      {
       RCALC.ShowError((IntPtr)0);
       Repcoder.exit();
       return;
      }

// Show compiled text in the Console
    string compstr = rc.CompiledText(true);
    if( compstr != String.Empty )
      {
       Console.WriteLine("\n******* Compiled text **************\n");
       Console.WriteLine(compstr);
       Console.WriteLine("\n******* End of compiled text ********\n");
      }

// Get and print the number of output values
    int nOut = rc.OutputCount();
    Console.WriteLine("nOut = " + nOut);

// Execute with some input values: x = 3.14159, y = 2.71
    ok = rc.Execute("3.14159###2.71");
    if( !ok )
      {
       RCALC.ShowError((IntPtr)0);
       Repcoder.exit();
       return;
      }

// Print the results
    Console.WriteLine("Out1 = " + rc.OutString(1));
    Console.WriteLine("Out2 = " + rc.OutString(2) + "\n");

// Get the results to host variables
```

```
    double Out1 = rc.OutDouble(1);
    double Out2 = rc.OutDouble(2);
    Console.WriteLine("Out1 = " + Out1 + ", Out2 = " + Out2);

// Exit repcoder
    Repcoder.exit();
   }
}
```

23. A sample application in Java

For Windows Java programmers a simple class RCALC was defined in the **Repcoder.java** file. It allows to call the exported functions of **repcoder.dll**. The Java class is very similar to the C# class. But there is no destructor here. Use the Free() method instead.

```
class RCALC
{
int Handle;

public RCALC();
public RCALC(RCALC rc);
public boolean Free();
public boolean CopyFrom(RCALC rc);

public int  GetHandle();
public int  GetState();
public boolean IsValid();
public boolean IsCompiled();
public boolean IsExecuted();

public void SetStatusWindow(long hStatWnd);

public boolean Compile(String Source);
public boolean CompileFile(String FileName, int Format);

public boolean Execute(String[] InputValues);
public boolean Execute(String InputValues, String Separator);
public boolean Execute(String InputValues);
public boolean Execute();

public String CompiledText(boolean bCRLF);

public int InputCount();
public int InputType(int number);
public int InputTypeNumericScale(int number);
public String InputTypeName(int number);
public String InputName(int number);
public boolean InputIsNull(int number);
public String InputString(int number);
public double InputDouble(int number);
public int InputInt(int number);
public long InputNumeric(int number);
public long InputTime(int number);

public int OutputCount();
public int OutType(int number);
public int OutTypeTextLen(int number);
```

```
public int OutTypeNumericScale(int number);
public String OutTypeName(int number);
public String OutName(int number);
public boolean OutIsNull(int number);
public String OutString(int number);
public double OutDouble(int number);
public int OutInt(int number);
public long OutNumeric(int number);
public long OutTime(int number);

public static int Year(long value);
public static int Month(long value);
public static int Day(long value);
public static int Hour(long value);
public static int Minute(long value);
public static int Second(long value);
public static int Millisec(long value);

public static boolean LastError();
public static void ShowError(long hParent);
public static String ErrorString();
}
```

A simple JAVA example:

```
//compile: javac -cp jna.jar Repcoder.java Test.java
//run: java -cp jna.jar; Test

public class Test
{
 public static void main(String[] args)
   {
// Init repcoder
    if( !Repcoder.init() )
      {
       System.out.println("Repcoder.init() error ...");
       return;
      }

// Create RCALC object
    RCALC rc = new RCALC();

// Text of an algorithm
    String AlgTxt =
    "INPUT {" +
    "FLOAT x;" +
    "FLOAT y;" +
    "}" +
    "FLOAT z = x + y;" +
    "OUTPUT {" +
    "FLOAT Out1 = z*z;" +
    "FLOAT Out2 = sqrt(x + y);" +
    "}";

// Compile RCALC object
    boolean ok = rc.Compile(AlgTxt);
    if( !ok )
      {
       RCALC.ShowError(0);
       Repcoder.exit();
       return;
```

```
      }

// Show compiled text in the Console
    String compstr = rc.CompiledText(true);
    if( !compstr.isEmpty() )
      {
       System.out.println("\n******* Compiled text **************\n");
       System.out.println(compstr);
       System.out.println("\n******* End of compiled text ********\n");
      }

// Get and print the number of output values
    int nOut = rc.OutputCount();
    System.out.println("nOut = " + nOut);

// Execute with some input values: x = 3.14159, y = 2.71
    ok = rc.Execute("3.14159###2.71");
    if( !ok )
      {
       RCALC.ShowError((IntPtr)0);
       Repcoder.exit();
       return;
      }

// Print the results
    System.out.println("Out1 = " + rc.OutString(1));
    System.out.println("Out2 = " + rc.OutString(2) + "\n");

// Get the results to host variables
    double Out1 = rc.OutDouble(1);
    double Out2 = rc.OutDouble(2);
    System.out.println("Out1 = " + Out1 + ", Out2 = " + Out2);

// Call Free() to clear resources !!!
    rc.Free();

// Exit repcoder
    Repcoder.exit();
   }
}
```

24. A sample application in Delphi

For Delphi programmers a simple class RCALC was defined in the **repcoder.pas** file. It allows to call the exported functions of **repcoder.dll**. The Delphi class is very similar to the C# class.

```
type
  RCALC = class

  private
  Handle: Integer;

  public

  constructor Create; overload;
  constructor Create(rc: RCALC); overload;
```

```
  destructor  Destroy; override;

  function CopyFrom(rc: RCALC): Boolean;
  function GetHandle: Integer;
  function GetState: Integer;
  function IsValid: Boolean;
  function IsCompiled: Boolean;
  function IsExecuted: Boolean;
  procedure SetStatusWindow(hStatWnd: HWND);

  function Compile(Source: WideString): Boolean; overload;
  function Compile(Source: AnsiString): Boolean; overload;
  function CompileFile(Source: WideString; Format: Integer): Boolean;
overload;
  function CompileFile(Source: AnsiString; Format: Integer): Boolean;
overload;

  function Execute(nInput: Integer; pInputValues: PWideString): Boolean;
overload;
  function ExecuteA(nInput: Integer; pInputValues: PAnsiString): Boolean;
overload;
  function Execute(InputValues: WideString; Separator: WideString):
Boolean; overload;
  function ExecuteA(InputValues: AnsiString; Separator: AnsiString):
Boolean; overload;
  function Execute(InputValues: WideString): Boolean; overload;
  function ExecuteA(InputValues: AnsiString): Boolean; overload;
  function Execute: Boolean; overload;

  function CompiledText(bCRLF: Boolean): WideString;
  function CompiledTextA(bCRLF: Boolean): AnsiString;

  function InputCount: Integer;
  function InputType(number: Integer): Integer;
  function InputTypeNumericScale(number: Integer): Integer;
  function InputTypeName(number: Integer): WideString;
  function InputTypeNameA(number: Integer): AnsiString;
  function InputName(number: Integer): WideString;
  function InputNameA(number: Integer): AnsiString;
  function InputIsNull(number: Integer): Boolean;
  function InputString(number: Integer): WideString;
  function InputStringA(number: Integer): AnsiString;
  function InputDouble(number: Integer): Double;
  function InputInt(number: Integer): Integer;
  function InputNumeric(number: Integer): Int64;
  function InputTime(number: Integer): Int64;

  function OutputCount: Integer;
  function OutType(number: Integer): Integer;
  function OutTypeTextLen(number: Integer): Integer;
  function OutTypeNumericScale(number: Integer): Integer;
  function OutTypeName(number: Integer): WideString;
  function OutTypeNameA(number: Integer): AnsiString;
  function OutName(number: Integer): WideString;
  function OutNameA(number: Integer): AnsiString;
  function OutIsNull(number: Integer): Boolean;
  function OutString(number: Integer): WideString;
  function OutStringA(number: Integer): AnsiString;
  function OutDouble(number: Integer): Double;
  function OutInt(number: Integer): Integer;
  function OutNumeric(number: Integer): Int64;
```

```
  function OutTime(number: Integer): Int64;
end;
//------------------

// RCALC global functions:
function RCALC_Year(value: Int64): Integer;
function RCALC_Month(value: Int64): Integer;
function RCALC_Day(value: Int64): Integer;
function RCALC_Hour(value: Int64): Integer;
function RCALC_Minute(value: Int64): Integer;
function RCALC_Second(value: Int64): Integer;
function RCALC_Millisec(value: Int64): Integer;
function RCALC_LastError: Boolean;
procedure RCALC_ShowError(hParent: HWND);
function RCALC_ErrorString: WideString;
function RCALC_ErrorStringA: AnsiString;
```

A simple Delphi example:

```
// compile: dcc32 -CC test.pas repcoder.pas

program test;

uses repcoder;

var
rc: RCALC;
bOk: Boolean;
nOut: Integer;
Out1: Double;
Out2: Double;
AlgTxt: AnsiString;

begin

// Init repcoder
if repc_init = False then
begin
  WriteLn('repc_init() error ...');
  Halt(1);
end;

// Alloc RCALC handle
rc := RCALC.Create;

// Text of an algorithm
AlgTxt :=
'INPUT {' +
'FLOAT x;' +
'FLOAT y;' +
'}' +
'FLOAT z = x + y;' +
'OUTPUT {' +
'FLOAT Out1 = z*z;' +
'FLOAT Out2 = sqrt(x + y);' +
'}';

// Compile RCALC handle
bOk := rc.Compile(AlgTxt);
```

```
if bOk = False then
begin
  RCALC_ShowError(0); // Show error in a MessageBox
  repc_exit;
  Halt(1);
end;

// Show compiled text in the Console
WriteLn(#13#10 + '******* Compiled text ***************' + #13#10);
WriteLn(rc.CompiledText(True));
WriteLn(#13#10 + '******* End of compiled text ********' + #13#10);

// Get and print the number of output values
nOut := rc.OutputCount;
WriteLn('nOut = ', nOut);

// Execute with some input values: x = 3.14159, y = 2.71
bOk := rc.Execute('3.14159###2.71');
if bOk = False then
begin
  RCALC_ShowError(0);
  repc_exit;
  Halt(1);
end;

// Print the results
WriteLn('Out1 = ', rc.OutString(1));
WriteLn('Out2 = ', rc.OutString(2) + #13#10);

// Get the results to host variables
Out1 := rc.OutDouble(1);
Out2 := rc.OutDouble(2);
WriteLn('Out1 = ', Out1, ', Out2 = ', Out2);

// Exit repcoder
repc_exit;

end.
```

25. The RCALC compiler and thread-safety

The RCALC API (in REPCoder.DLL) provides exported functions that allow the calling application to compile and execute the script code in the runtime. You can compile it once (to a binary resource in memory) and then execute many times with different sets of input values. In this sense RCALC is not an interpreted but rather a compiled language. The API functions access the created, compiled and executed binaries through a 32-bit integer RCALC **handle**. These functions allow to extract information about the input and output variables (names and types), entered input values and calculated output results. There are also functions returning error messages.

The RCALC API is thread-safe and prepared for multithreading. It means that you can execute the compiled code in different threads at the same time. But a handle must always be owned by one thread only. The owner thread is the one that created the handle (with the rcalc_alloc_handle() function). Only the owner thread of a handle can call API functions on this handle. Otherwise an error is generated with the message:

'This RCALC handle belongs to another thread'

So you can simply do all the job in a separate thread: create, compile, execute, get results, free.

create → compile → execute → get results → free

But it would be a waste of computer resources to compile the same code many times in different threads. A better practice is first to create and compile a primary handle in a primary thread (the program main function). Then you can make a copy of this compiled handle inside the working procedure of another thread, where it will be finally executed (once or many times in a loop). The important API function to copy the handle across different threads is rcalc_copy_handle(). The returned new handle is a copy (all resources are duplicated) of the source handle and is owned by a thread that called the copy function. The source handle remains of course unchanged and still belongs to the source thread. At the end each handle should be freed with rcalc_free_handle(). Only the owner thread of a handle can call this function with success.

26. A multi-thread example in C

The below C example shows how to use the RCALC API in a multi-thread application. Here we want to execute some RCALC algorithm simultaneously in 11 different threads. The text of the algorithm is in the external file `threads.txt`. The INPUT section accepts 2 arguments: `thrnum` (thread number: 1,2,…,11), `nloop` (number of loops). The OUTPUT section is empty. The script creates a result file: `res_1.txt` or `res_2.txt` or … `res_11.txt`, depending on the thread number, where it logs the progress of the algorithm execution. The algorithm runs 2 WHILE loops. In the first loop it adds many (nloop) times a single `'a'` character to the `s1` TEXT variable using the + operator. In the second loop the much faster += operator is used to add a 2-character text `'aa'`. The times of execution of the loops are measured (`Time1`, `Time2`) and also written to the result file:

```
INPUT {
INT thrnum;
INT nloop;
}

INT ok = 0;
TEXT NL = tochar(13) + tochar(10);
TEXT snum = totext(thrnum);
TEXT fname = 'res_' + snum + '.txt';
ok = writetext(fname, 0, NL + 'Thread number: ' + snum + NL + NL);
INT m = nloop;
IF( m > 300000 )
  {
   m = 300000;
   ok = writetext(fname, 1, 'Input nloop truncated to: 300000 ...' + NL);
  }

TIME t1 = currtime();
```

```
TEXT s1 = '';
INT i = 0;
WHILE(i < m)
  {
   s1 = s1 + 'a';
   i += 1;

   IF( imod(i, 10000) == 0 )
     {
      ok = writetext(fname, 1, 'i = ' + totext(i) + NL);
     }
  }

TIME t2 = currtime();
//----------------

ok = writetext(fname, 1, NL + 'And now faster. We use operator += to expand
text:' + NL + NL);

TEXT s2 = '';
i = 0;
WHILE( i < m )
  {
   s2 += 'aa';
   i += 1;

   IF( imod(i, 10000) == 0 )
     {
      ok = writetext(fname, 1, 'i = ' + totext(i) + NL);
     }
  }

TIME t3 = currtime();
//----------------

INT N = m;
INT L1 = len(s1);
FLOAT Time1 = t2-t1;
INT L2 = len(s2);
NUMERIC(3) Time2 = t3-t2;
TEXT res = NL + NL;
res += 'N = ' + totext(N) + NL;
res += 'L1 = ' + totext(L1) + NL;
res += 'Time1 = ' + totext(Time1) + NL;
res += 'L2 = ' + totext(L2) + NL;
res += 'Time2 = ' + totext(Time2) + NL;

ok = writetext(fname, 1, res);

OUTPUT {
}
```

The code of the program first creates the primary handle in the main function an compiles it. This handle is passed to the thread procedure of each child thread. Each thread makes its own copy of the handle calling the rcalc_copy_handle() function and executes the RCALC algorithm code. This way the RCALC code is compiled only once in the program. Each thread must also free its own handle after it is used.

```
/*
compile VC++: cl /EHsc thrtest1.cpp repcoder.c user32.lib
```

```
compile BCC32: bcc32 -tWM thrtest.cpp repcoder.c
*/

#include "repcoder.h"
#include <process.h>
#include <stdio.h>
//------------------------

static const int NUM_THREADS = 10; // number of threads
static const int N_LOOP = 50000; // number of WHILE loops

// Input structure for a thread
struct THR_STRUCT
{
 int Nr;
 int CalcHandle;
 volatile int bEnd;
};

// Thread procedure
static void _ThrProc(void* ptr)
{
if( !ptr ) return;
THR_STRUCT* pts = (THR_STRUCT*)ptr;

// Make a copy of the primary (compiled) handle
int h = rcalc_copy_handle(pts->CalcHandle);
if( !h ) { rcalc_show_error(0); pts->bEnd = 1; return; }

// Set the 2 input values for the RCALC script
wchar_t InputValues[100];
wsprintfW(InputValues, L"%d###%d", pts->Nr, N_LOOP);

// Execute the RCALC code
rcalc_execute2(h, InputValues, 0);

// Free the copied handle
rcalc_free_handle(h);

// Show RCALC error (if any)
rcalc_show_error(0);

// Mark the volatile flag that the thread has finished
// for the join code in the main() function
pts->bEnd = 1;
}
//------------------------

int main()
{
// Load and init repcoder.dll
if( !repcoder_load() ) return 0;
if( !repc_init() ) { repcoder_free(); return 0; }

// Alloc the primary RCALC handle in the main thread
int h = rcalc_alloc_handle();
if( !h ) { repc_exit(); repcoder_free(); return 0; }

// Compile the handle (only once !!!)
BOOL ok = rcalc_compile_file(h, L"threads.txt", 0);
if( !ok )
```

```
  {
   rcalc_show_error(0);

   rcalc_free_handle(h);
   repc_exit();
   repcoder_free();
   return 0;
  }

// Allocate input structures for child threads
int nThr = NUM_THREADS;
THR_STRUCT* params = new THR_STRUCT[nThr];
if( !params )
  {
   rcalc_free_handle(h);
   repc_exit();
   repcoder_free();
   return 0;
  }

// Start the child threads
for(int i=0; i<nThr; i++)
  {
   params[i].Nr = i+1;
   params[i].CalcHandle = h;
   params[i].bEnd = 0;
   _beginthread(_ThrProc, 0, params+i);
  }

// Execute the RCALC code also in the main thread:
// ... make also a copy of the primary (compiled) handle
int h2 = rcalc_copy_handle(h);
// ... set the 2 input values for the RCALC script
wchar_t InputValues[100];
wsprintfW(InputValues, L"%d###%d", nThr+1, N_LOOP);
// Execute the RCALC code
rcalc_execute2(h2, InputValues, 0);
// Free the copied handle
rcalc_free_handle(h2);

// Join the threads
while(1)
  {
   int bStop = 1;
   for(int i=0; i<nThr; i++)
     {
      if( params[i].bEnd == 0 ) { bStop = 0; break; }
     }
   if( bStop ) break;
   Sleep(100);
  }

// Delete the input structures
delete[] params;

// Free the primary handle
rcalc_free_handle(h);

// Show RCALC error (if any)
rcalc_show_error(0);
```

```
// Exit and free repcoder.dll
repc_exit();
repcoder_free();
return 0;
}
```

27. A multi-thread example in C++

The equivalent C++ code is using the RCALC class instead of the integer handle. You also do not need to free the RCALC handles because the destructors do this job.

```
/*
compile VC++: cl /EHsc thrtest2.cpp rcalc_class.cpp repcoder.c user32.lib
compile BCC32: bcc32 -tWM thrtest2.cpp rcalc_class.cpp repcoder.c
*/

#include "repcoder.h"
#include "rcalc_class.h"
#include <process.h>
//------------------------

static const int NUM_THREADS = 10; // number of threads
static const int N_LOOP = 50000; // number of WHILE loops

// Input structure for a thread
struct THR_STRUCT
{
int Nr;
RCALC* pCalc;
volatile int bEnd;
};

// Thread procedure
static void _ThrProc(void* ptr)
{
if( !ptr ) return;
THR_STRUCT* pts = (THR_STRUCT*)ptr;

// Make a copy of the primary (compiled) object
RCALC rc( *(pts->pCalc) );
if( !rc.IsValid() ) { RCALC::ShowError(); pts->bEnd = 1; return; }

// Set the 2 input values for the RCALC script
char InputValues[100];
wsprintfA(InputValues, "%d###%d", pts->Nr, N_LOOP);

// Execute the RCALC code
rc.Execute(InputValues);

// Show RCALC error (if any)
RCALC::ShowError();

// Mark the volatile flag that the thread has finished
// for the join code in the main() function
pts->bEnd = 1;
}
//------------------------
```

```
int main()
{
// Load and init repcoder.dll
if( !repcoder_load() ) return 0;
if( !repc_init() ) { repcoder_free(); return 0; }

// Create the primary RCALC object in the main thread
RCALC rc;
if( !rc.IsValid() ) { repc_exit(); repcoder_free(); return 0; }

// Compile the object (only once !!!)
bool ok = rc.CompileFile("threads.txt");
if( !ok )
  {
   RCALC::ShowError();

   repc_exit();
   repcoder_free();
   return 0;
  }

// Allocate input structures for child threads
int nThr = NUM_THREADS;
THR_STRUCT* params = new THR_STRUCT[nThr];
if( !params )
  {
   repc_exit();
   repcoder_free();
   return 0;
  }

// Start the child threads
for(int i=0; i<nThr; i++)
  {
   params[i].Nr = i+1;
   params[i].pCalc = &rc;
   params[i].bEnd = 0;
   _beginthread(_ThrProc, 0, params+i);
  }

// Execute the RCALC code also in the main thread:
// ... make also a copy of the primary (compiled) object
RCALC rc2(rc);
// ... set the 2 input values for the RCALC script
char InputValues[100];
wsprintfA(InputValues, "%d###%d", nThr+1, N_LOOP);
// Execute the RCALC code
rc2.Execute(InputValues);

// Join the threads
while(1)
  {
   int bStop = 1;
   for(int i=0; i<nThr; i++)
     {
      if( params[i].bEnd == 0 ) { bStop = 0; break; }
     }
   if( bStop ) break;
   Sleep(100);
  }
```

```
// Delete the input structures
delete[] params;

// Show RCALC error (if any)
RCALC::ShowError();

// Exit and free repcoder.dll
repc_exit();
repcoder_free();
return 0;
}
```

28. A multi-thread example in C#

The equivalent C# code is very similar to the C++ code. A special class `MyThrParam` is used to pass the parameters to the static thread procedure.

```
//compile: csc thrtest.cs repcoder.cs

using System;
using System.Threading;

class MyThrParam
{
public int nr;
public RCALC rc;

public MyThrParam(int a_nr, RCALC a_rc)
  {
   nr = a_nr;
   rc = a_rc;
  }
}
//--------------

class App
{
 const int NUM_THREADS = 10; // number of threads
 const int N_LOOP = 50000; // number of WHILE loops

// Thread procedure
 public static void ThrProc(object data)
   {
    MyThrParam par = (MyThrParam)data;

// Make a copy of the primary (compiled) object
    RCALC rc = new RCALC(par.rc);
    if( !rc.IsValid() ) { RCALC.ShowError((IntPtr)0); return; }

// Set the 2 input values for the RCALC script
    string InputValues = par.nr + "###" + N_LOOP;

// Execute the RCALC code
    rc.Execute(InputValues);

// Show RCALC error (if any)
```

```
    RCALC.ShowError((IntPtr)0);
   }

 public static void Main()
   {
// Init repcoder
    if( !Repcoder.init() )
      {
       Console.WriteLine("Repcoder.init() error ...");
       return;
      }

// Create the primary RCALC object in the main thread
    RCALC rc = new RCALC();

// Compile the object (only once !!!)
    bool ok = rc.CompileFile("threads.txt", 0);
    if( !ok )
      {
       RCALC.ShowError((IntPtr)0);
       Repcoder.exit();
       return;
      }

// Allocate input structures for child threads
    MyThrParam[] myparams = new MyThrParam[NUM_THREADS];
    Thread[] threads = new Thread[NUM_THREADS];

// Start the child threads
    for(int i=0; i<NUM_THREADS; i++)
      {
       myparams[i] = new MyThrParam(i+1, rc);
       threads[i] = new Thread(App.ThrProc);
       threads[i].Start(myparams[i]);
      }

// Execute the RCALC code also in the main thread:
// ... make also a copy of the primary (compiled) object
    RCALC rc2 = new RCALC(rc);
// ... set the 2 input values for the RCALC script
    string InputValues = (NUM_THREADS+1) + "###" + N_LOOP;
// Execute the RCALC code
    rc2.Execute(InputValues);

// Join the threads
    for(int i=0; i<NUM_THREADS; i++) threads[i].Join();

// Show RCALC error (if any)
    RCALC.ShowError((IntPtr)0);

// Exit repcoder
    Repcoder.exit();
  }
}
```