The module of subsystem “Data acquisition” <JavaLikeCalc>

Module:	JavaLikeCalc
Name:	Calculator based on Java-like language.
Type:	DAQ
Source:	daq_JavaLikeCalc.so
Version:	1.9.5
Author:	Roman Savochenko
Translated:	Maxim Lysenko
Description:	Provides based on java like language calculator and engine of libraries. The user can create and modify functions and libraries.
License:	GPL

Contents table

Introduction

The module of controller *JavaLikeCalc* provides a mechanism for creating of functions and libraries on Java-like language. Description of functions on Java-like language is reduced to the binding parameters of the function with algorithm. In addition, the module has the functions of the direct computation by creation of the computing controllers.

Direct computations are provided by the creation of controller and linking it with the function of this module. For linked function it is created the frame of values, with which the periodically calculating is carried out.

The module implements the functions of the horizontal redundancy, namely, working in conjunction with the remote station of the same level. In addition to the synchronization of the archives of values and archives of attributes of parameters the module implements synchronization of computational function, in order to shockless catch of the algorithms.

Parameters of functions can be freely created, deleted or modified. The current version of the module supports up to 65535 parameters of the function in the sum with the internal variables. View of the editor of functions is shown in Figure 1.

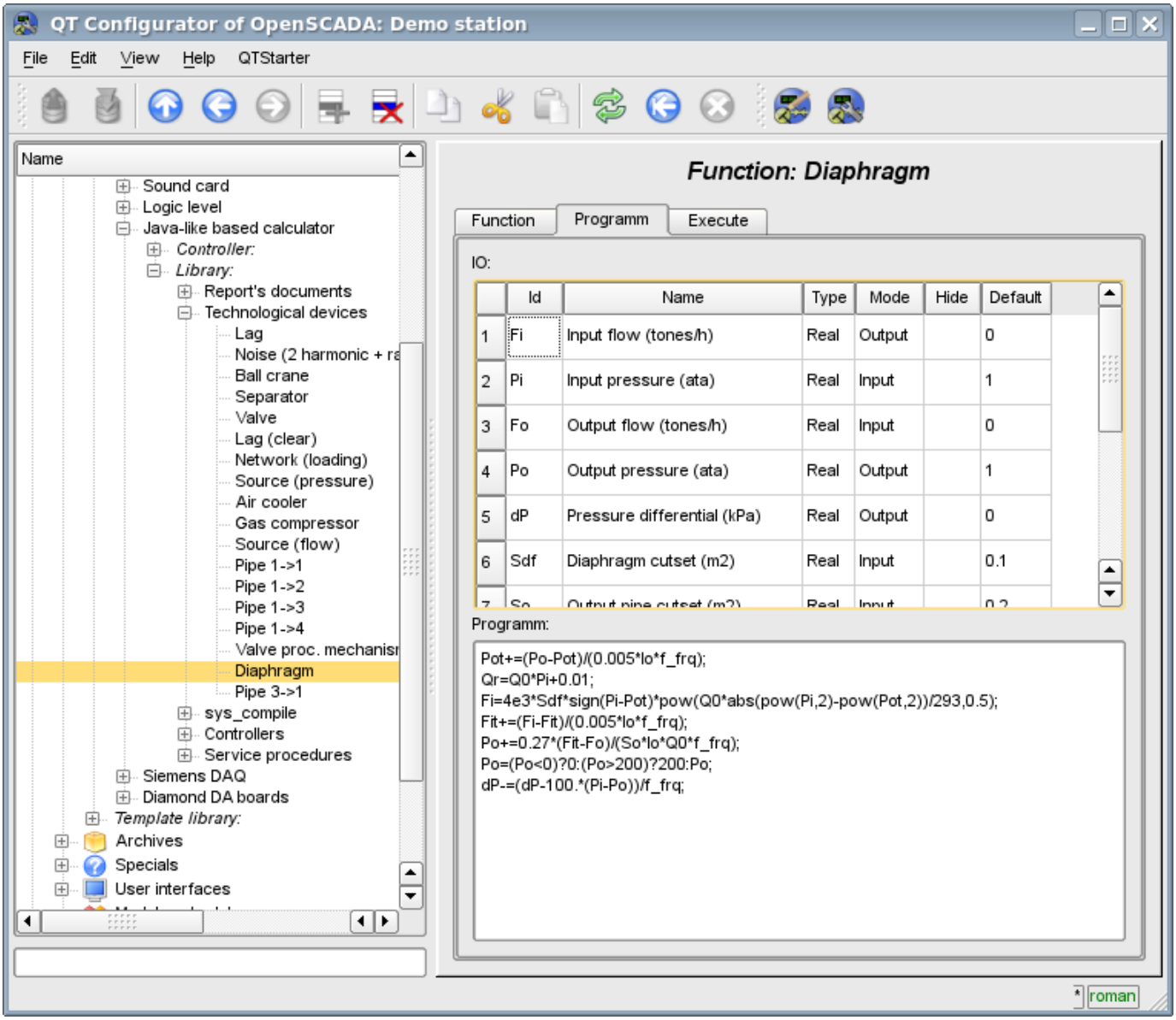

Fig.1. View of the editor of functions.

After any program changing or configuration of parameters recompiling of the programs with forestalling of linked with function objects of values of TValCfg is performed. Language compiler is built

using well-known generator grammar «Bison», which is compatible with the not less well-known tool Yacc.

The language uses the implicit definition of local variables, which is to define a new variable in the case of assigning a value to it. This type of local variable is set according to the type of the assigning value. For example, the expression <Qr=Q0*Pi+0.01;> will define Qr variable with the type of variable Q0.

In working with various types of data language uses the mechanism of casting the types in the places where such casting is appropriate.

To comment the sections of code in the language it is provided «//» and «/ * ... * /» characters. Everything that comes after "//" up to the end of the line and between «/ * ... * /», is ignored by the compiler.

During the code generation language compiler produces an optimization of constants and casting the types of the constants to the required type. Optimizing of the constants means the implementation of computing of the constants in the process of building of the code under the two constants and paste the result in the code. For example, the expression <y=pi*10;> reduces to a simple assignment <y=31.4159;>. Casting the types of constants to the required type means formation of the constant in the code which excludes the cast in the execution process. For example, the expression <y=x*10>, in the case of a real type of the variable x, is transformed into <y=x*10.0>.

The language supports calls of the external and internal functions. Name of any function in general is perceived as a character, test for ownership of which by a particular category is done in the following order:

- keywords;
- constants;
- built-in functions;
- external functions, object's functions and OpenSCADA nodes functions (DOM) ;
- already registered characters of variables, object's attributes and hierarchy of objects DOM;
- new attributes of the system parameters;
- new function parameters;
- new automatic variable.

Call of the external function, attribute of system parameters is written as an address to the object of dynamic tree of the object model of the system OpenSCADA in the form of: <DAQ.JavaLikeCalc.lib_techApp.klapNotLin>.

To provide the possibility of writing custom procedures for the administration of the various components of OpenSCADA module provides the implementation of API pre-compilation of custom procedures of individual components of OpenSCADA on the implementation of Java-like language. These components are already: Templates of the parameters of subsystem “Data acquisition” and Visual control area (VCA).

1. Java-like language

1.1. Elements of language

Keywords: if, else, while, for, break, continue, return, using, true, false.

Constants:

- decimal: numerals 0–9 (12, 111, 678);
- octal: numerals 0–7 (012, 011, 076);
- hexadecimal: numerals 0–9, letters a-f or A-F (0x12, 0XAB);
- real: 345.23, 2.1e5, 3.4E-5, 3e6;
- boolean: true, false;
- string: "hello", without next string went but with support of a direct concatenation of string constants.

Types of variables:

- integer: $-2^{31}...2^{31}$, EVAL_INT(-2147483647);
- real: $3.4 * 10^{308}$, EVAL_REAL(-3.3E308);
- boolean: false, true, EVAL_BOOL(2);
- string: sequence of characters-bytes (0...255) any length, limited by memory capacity and DB storage; EVAL_STR("<EVAL>").

Built-in constants: pi = 3.14159265, e = 2.71828182, EVAL_BOOL(2), EVAL_INT(-2147483647), EVAL_REAL(-3.3E308), EVAL_STR("<EVAL>")

Attributes of the parameters of system OpenSCADA (starting from subsystem DAQ, as follows <Type of DAQ module>.< Controller>.<Parameter>.<Attribute>).

The functions of the object model of the system OpenSCADA.

1.2. Operations of language

Operations supported by the language presented in the table below. The priority of operations is reduced from top to bottom. Operations with the same priority is composed of one color group.

Symbol	Описание
()	Call of the function.
{}	Program blocks.
++	Increment (post and pre).
--	Decrement (post and pre).
-	Unary minus.
!	Logical negation.
~	Bitwise negation.
*	Multiplication.
/	Division.
%	The remainder of integer division.
+	Addition
-	Subtraction
<<	Bitwise shift left
>>	Bitwise shift right
>	Greater
>=	Greater than or equal to

Symbol	Описание
<	Less
<=	Less than or equal to
==	Equals
!=	Unequal
|	Bitwise «OR»
&	Bitwise «AND»
^	Bitwise «Exclusive OR»
&&	Boolean «AND»
||	Boolean «OR»
?:	Conditional operation (i=(i<0)?0:i;)
=	Assignment.
+=	Assignment with addition.
-=	Assignment with subtraction.
*=	Assignment with multiplication.
/=	Assignment with division.

1.3. Embedded functions of language

Virtual machine of the language provides the following set of built-in functions general-purpose:

- double max(double x, double x1) — maximum value of *x* and *x1*;
- double min(double x, double x1) — minimum value of *x* and *x1*;
- string typeof(ElTp vl) — type of value *vl*.

To ensure a high speed in mathematical calculations module provides embedded mathematical functions that are called at the level of commands of virtual machine. Predefined mathematical functions:

- double sin(double x) — sine *x*;
- double cos(double x) — cosine *x*;
- double tan(double x) — tangent *x*;
- double sinh(double x) — hyperbolic sine of *x*;
- double cosh(double x) — hyperbolic cosine of *x*;
- double tanh(double x) — hyperbolic tangent of *x*;
- double asin(double x) — arcsine of *x*;
- double acos(double x) — arc cosine of *x*;
- double atan(double x) — arctangent of *x*;
- double rand(double x) — random number from 0 to *x*;
- double lg(double x) — decimal logarithm of *x*;
- double ln(double x) — natural logarithm of *x*;
- double exp(double x) — exponent of *x*;
- double pow(double x, double x1) — erection of x to the power *x1*;
- double sqrt(double x) — the square root of *x*;
- double abs(double x) — absolute value of *x*;
- double sign(double x) — sign of *x*;
- double ceil(double x) — rounding the number *x* to a greater integer;
- double floor(double x) — rounding the number *x* to a smaller integer.

1.4. Operators of the language

The total list of operators of the language:

- *var* — operator for variable initialize;
- *if* — operator of the condition "If";
- *else* — operator of the condition "ELSE";
- *while* — description of the loop while;
- *for* — description of the loop for;
- *in* — for-cycle separator for object's properties scan;
- *break* — interrupt of the execution of the loop;
- *continue* — continue the execution of the loop from the beginning;
- *using* — allows to establish scope of functions of often used library (using Special.FLibSYS;) for future reference only by means of the function name;
- *return* — interruption of the function and return of the result, the result is copied to the attribute with the flag return (return 123;);
- *new* — object creation, realized object "Object", massif "Array" and regular expressions "RegExp".

1.4.1. Conditional operators

The language of module supports two types of conditions. First — this is the operation of condition for use within the expression, the second — a global, based on the conditional operators.

Conditions inside the expression is based on the operations of «?» And «:». As an example we'll write the following practical expression <st_open=(pos>=100)?true:false;>, which reads as «If the variable <*pos*> greater than or equal to 100, the variable st_open is set to true, otherwise — to false.

The global condition is based on the conditional operators «if» and «else». An example is the same expression, but written by other means <if(pos>100) st_open=true; else st_open=false;>. As shown, the expression is written in a different way, but is read in the same way.

1.4.2. Loops

Two types of loops are supported: while, for and for-in. The syntax of the loops corresponds to programming languages: C++, Java, and JavaScript.

Loop **while** generally written as follows:

while(<condition>) <body of the loop>;

Loop **for** is written as follows:

for(<pre-initialization>;<condition>;<post-calculation>) <body of the loop>;

Loop **for-in** is written as follows:

for(<variable> in <object>) <body of the loop>;

Where:

<*condition*> — expression, determining the condition;
<*body of the loop*> — the body of the loop of multiple execution;
<*pre-initialization*> — expression of pre-initialization of variable of the loop;
<*post-calculation*> — expression of modification of parameters of the loop after the next iteration;
<*variable*> — variable, which will contain object's properties name at scan;
<*object*> — object for which properties scan gone.

1.4.3. Special characters of string variables

The language supports the following special characters of string variables:

"\n" — line feed;
"\t" — tabulation symbol;
"\b" — culling;

"\f" — page feed;
"\r" — carriage return;
"\\" — the character itself '\'.
"\041" — symbol '!' wrote by octal number;
"\x21" — symbol '!' wrote by hex number.

1.5. Object

The language provides the data type "Object" support. The data type "Object" is associated container of properties and functions. The properties can support data of fourth basic types and other objects. The access to properties is doing through the dot to object <obj.prop> and also by property placement into the rectangle brackets <obj["prop"]>.It is obvious that the first mechanism is static, while the second lets you to specify the name of the property through a variable. Creating an object is carried by the keyword <new>: <varO = new Object()>. The basic definition of the object does not contain functions. Copying of an object is actually makes the reference to the original object. When you delete an object is carried out the reducing of the reference count, and when a reference count is set to zero then object is removed physically.

Different components can define basic object with special properties and functions. The standard extension of the object is an array "Array", which is created by the command <varO = new Array(prm1,prm2,prm3,...,prmN)>. Comma-separated parameters are placed in the array in the original order. If the parameter is the only one the array is initiated by the specified number of empty elements. Peculiarity of the array is that it works with the properties as the indexes and their complete naming is meaningless, and therefore the mechanism of addressing only by the placing the index into square brackets <arr[1]> is accessible. Array stores the properties in its own container of the one-dimensional array. Digital properties of the array are used to access directly to the array, and the characters work as object properties. For more details about the properties and functions of the array can be read here.

The object of regular expression RegExp is created by command *<varO = new RegExp(pat,flg)>*, where *<pat>* — pattern of regular expression, and *<flg>* — match flags. The object for work with regular expressions, based on the library PCRE. In the global search set object attribute "lastIndex", which allows you to continue searching for the next function call. In the case of an unsuccessful search for the attribute "lastIndex" reset to zero. For more details about the properties and functions of the array can be read here.

For random access to the arguments of the functions provided the arguments object, which you can refer to by the symbol "arguments". This object contains the property "length" with a number of arguments in functions and allows you to access to a value of the argument by its number or ID. Consider the enumeration of the arguments on the cycle:

```
args = new Array();
for(var i=0; i < arguments.length; i++)
      arg[i] = arguments[i];
```

The basic types have the partial properties of the object. Properties and functions of the basic types are listed below:

- NULL type, functions:
  - *bool isEVal();* — Return "true".
- Logical type, functions:
  - *bool isEVal();* — Check value to "EVAL".
  - *string toString();* — Performs the value as the string “true” or “false”.
- Integer and real number:
  Properties:
  - *MAX_VALUE* — maximum value;
  - *MIN_VALUE* — minimum value;
  - *NaN* — error value.

  Functions:
  - *bool isEVal();* — Check value to "EVAL".

- *string toExponential(int numbs = -1);* — Return the string of the number, formatted in exponential notation, and with the number of significant digits <*numbs*>. If <*numbs*> is missing the number of digits will have as much as needed.
- *string toFixed(int numbs = 0, int len = 0, bool sign = false);* — Return the string of the number, formatted in the notation of fixed-point, and with the number of significant digits after the decimal point <*numbs*> for minimum length <*len*> and strong sign present <*sign*>. If <*numbs*> is missing the number of digits after the decimal point is equal to zero.
- *string toPrecision(int prec = -1);* — Return the string of the formatted number with the number of significant digits <*prec*>.
- *string toString(int base = 10, int len = -1, bool sign = false);* — Return the string of the formatted number of integer type with the following representation base (2-36) for minimum length <*len*> and strong sign present <*sign*>.

- String:
  Properties:
  - *int length* — string length.

  Functions:
  - *bool isEVal();* — Check value to "EVAL".
  - *string charAt(int symb);* — Extracts from the string the symbol <*symb*>.
  - *int charCodeAt(int symb);* — Extracts from the string the symbol code <*symb*>.
  - *string concat(string val1, string val2, ...);* — Returns a new string formed by joining the values <*val1*> etc. to the original one.
  - *int indexOf(string substr, int start);* — Returns the position of the required string <*substr*> in the original row from the position <*start*>. If the initial position is not specified then the search starts from the beginning. If the search string is not found then -1 is returned.
  - *int lastIndexOf(string substr, int start);* — Returns the position of the search string <*substr*> in the original one beginning from the position of <*start*> when searching from the end. If the initial position is not specified then the search begins from the end. If the search string is not found then -1 is returned.
  - *int search(string pat, string flg = "");* — Search into the string by pattern <*pat*> and pattern's flags <*flg*>. Return found substring position or -1 for else.
    ```
    var rez = "Java123Script".search("script","i");
    // rez = 7
    ```
  - *int search(RegExp pat);* — Search into the string by RegExp pattern <*pat*>. Return found substring position or -1 for else.
    ```
    var rez = "Java123Script".search(new RegExp("script","i"));
    // rez = 7
    ```
  - *Array match(string pat, string flg = "");* — Call match for the string by pattern <*pat*> and flags <*flg*>. Return matched substring (0) and subexpressions (>0) array. Set "index" attribute of the array to substring position. Set "input" attribute to source string.
    ```
    var rez = "1 plus 2 plus 3".match("\\d+","g");
    // rez = [1], [2], [3]
    ```
  - *Array match(TRegExp pat);* — Call match for the string and RegExp pattern <*pat*>. Return matched substring (0) and subexpressions (>0) array. Set "index" attribute of the array to substring position. Set "input" attribute to source string.
    ```
    var rez = "1 plus 2 plus 3".match(new RegExp("\\d+","g"));
    // rez = [1], [2], [3]
    ```
  - *string slice(int beg, int end); string substring(int beg, int end);* — Return the string extracted from the original one starting from the <*beg*> position and ending be the <*end*>. If the beginning or end is negative, then the count is conducted from the end of the line. If the end is not specified, then the end is the end of the line.
  - *Array split(string sep, int limit);* — Return the array of strings separated by <*sep*> with the limit of the number of elements <*limit*>.
  - *Array split(RegExp pat, int limit);* — Return the array of strings separated by RegExp pattern <*pat*> with the limit of the number of elements <*limit*>.
    ```
    Rez = "1,2, 3 , 4 ,5".split(new RegExp("\\s*,\\s*"));
    ```

```
// rez = [1], [2], [3], [4], [5]
```

- *string insert(int pos, string substr);* — Insert substring <*substr*> into this string's position <*pos*>.
- *string replace(int pos, int n, string str);* — Replace substring into position <*pos*> and length <*n*> to string <*str*>.

```
rez = "Javascript".replace(4,3,"67");
// rez = "Java67ipt"
```

- *string replace(string substr, string str);* — Replace all substrings <*substr*> to string <*str*>.

```
Rez = "123 321".replace("3","55");
// rez = "1255 5521"
```

- *string replace(RegExp pat, string str);* — Replace substrings by pattern <*pat*> to string <*str*>.

```
rez = "value = \"123\"".replace(new
RegExp("\"([^\"]*)\"","g"),"``$1''"));
// rez = "value = ``123''"
```

- *real toReal();* — Convert this string to real number.
- *int toInt(int base = 0);* — Convert this string to integer number in accordance with the base <*base*> (from 2 to 36). If base is 0, then the prefix will be considered a record for determining the base (123-decimal; 0123-octal; 0x123-hex).
- *string parse(int pos, string sep = ".", int off = 0);* — Get token with numbet <*pos*> from the string when separated by <*sep*> and from offset <*off*>. Result offset is returned to back <*off*>.
- *string parsePath(int pos, int off = 0);* — Get path token with numbet <*pos*> from the string and from offset <*off*>. Result offset is returned to back <*off*>.
- *string path2sep(string sep = ".");* — Convert path into this string to separated by <*sep*> string.

For access to system objects (nodes) of the OpenSCADA the corresponding object is provided which is created simply by specifying the enter point "SYS" of the root object OpenSCADA, and then with the point separator the sub-objects in accordance with the hierarchy are specified. For example, the call of the request function over the outgoing transport is carried out as follows: *SYS.Transport.Sockets.out_testModBus.messIO(strEnc2Bin("15 01 00 00 00 06 01 03 00 00 00 05"));*.

1.6. Examples of programs on the language

Here are some examples of programs on Java-like language:

```
//Model of the course of the executive machinery of ball valve
if( !(st_close && !com) && !(st_open && com) )
{
      tmp_up=(pos>0&&pos<100)?0:(tmp_up>0&&lst_com==com)?tmp_up-1./frq:t_up;
      pos+=(tmp_up>0)?0:(100.*(com?1.:-1.))/(t_full*frq);
      pos=(pos>100)?100:(pos<0)?0:pos;
      st_open=(pos>=100)?true:false;
      st_close=(pos<=0)?true:false; lst_com=com;
}
//Valve model
Qr=Q0+Q0*Kpr*(Pi-1)+0.01;
Sr=(S_kl1*l_kl1+S_kl2*l_kl2)/100.;
Ftmp=(Pi>2.*Po)?Pi*pow(Q0*0.75/Ti,0.5):
      (Po>2.*Pi)?Po*pow(Q0*0.75/To,0.5):
      pow(abs(Q0*(pow(Pi,2)-pow(Po,2))/Ti),0.5);
Fi-=(Fi-7260.*Sr*sign(Pi-Po)*Ftmp)/(0.01*lo*frq);
Po+=0.27*(Fi-Fo)/(So*lo*Q0*frq);
Po=(Po<0)?0:(Po>100)?100:Po;
To+=(abs(Fi)*(Ti*pow(Po/Pi,0.02)-To)+
      (Fwind+1)*(Twind-To)/Riz)/(Ct*So*lo*Qr*frq);
```

2. Controller and its configuration

The controller of the module connects with the functions of libraries, built with his help, to provide immediate calculations. In order to provide calculated data in the system OpenSCADA parameters can be created in the controller. Example of the configuration tab of the controller of the given type depicted in Figure 2.

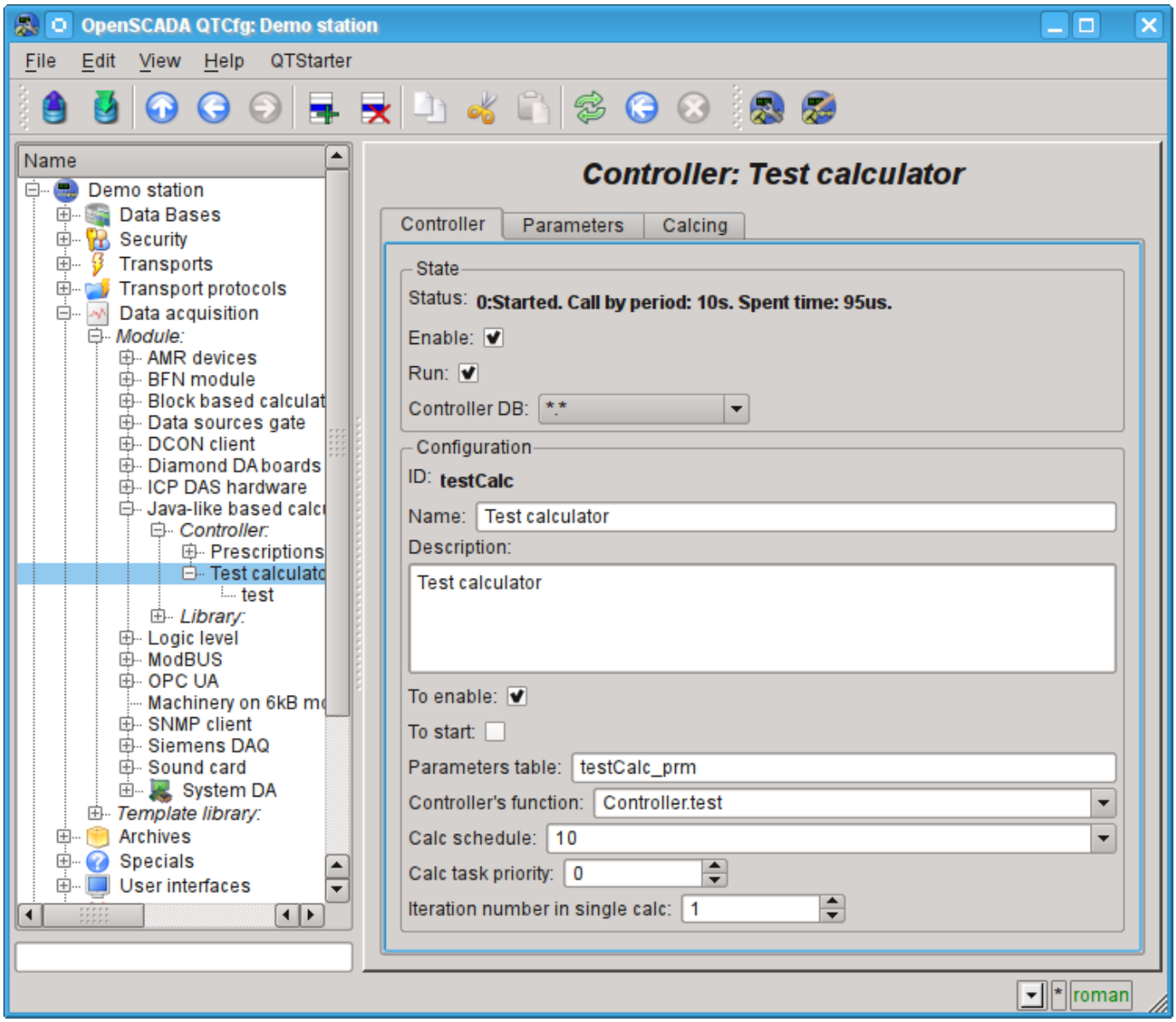

Fig.2. Configuration tab of the controller.

From this tab you can set:

- The state of the controller, as follows: Status, «Enable», «Run» and the name of the database containing the configuration.
- Id, name and description of the controller.
- The state, in which the controller must be translated at boot: «To enable» and «To start».
- Name of table to store the settings.
- Address of the computational function.
- The calculation schedule policy, priority and number of iterations in one cycle of calculating task.

Tab "Calculations" of the controller (Fig. 3) contains the parameters and the text of the program, directly performed by the controller. Also for monitoring of execution the time of calculating of the program is shown. Module provides a number of special options available in the controller program:

- *f_frq* — The controller program calculate frequency, read-only.
- *f_start* — First calculate of the controller program, start, read-only.
- *f_stop* — Last calculate of the controller program, stop, read-only.
- *this* — The controller object.

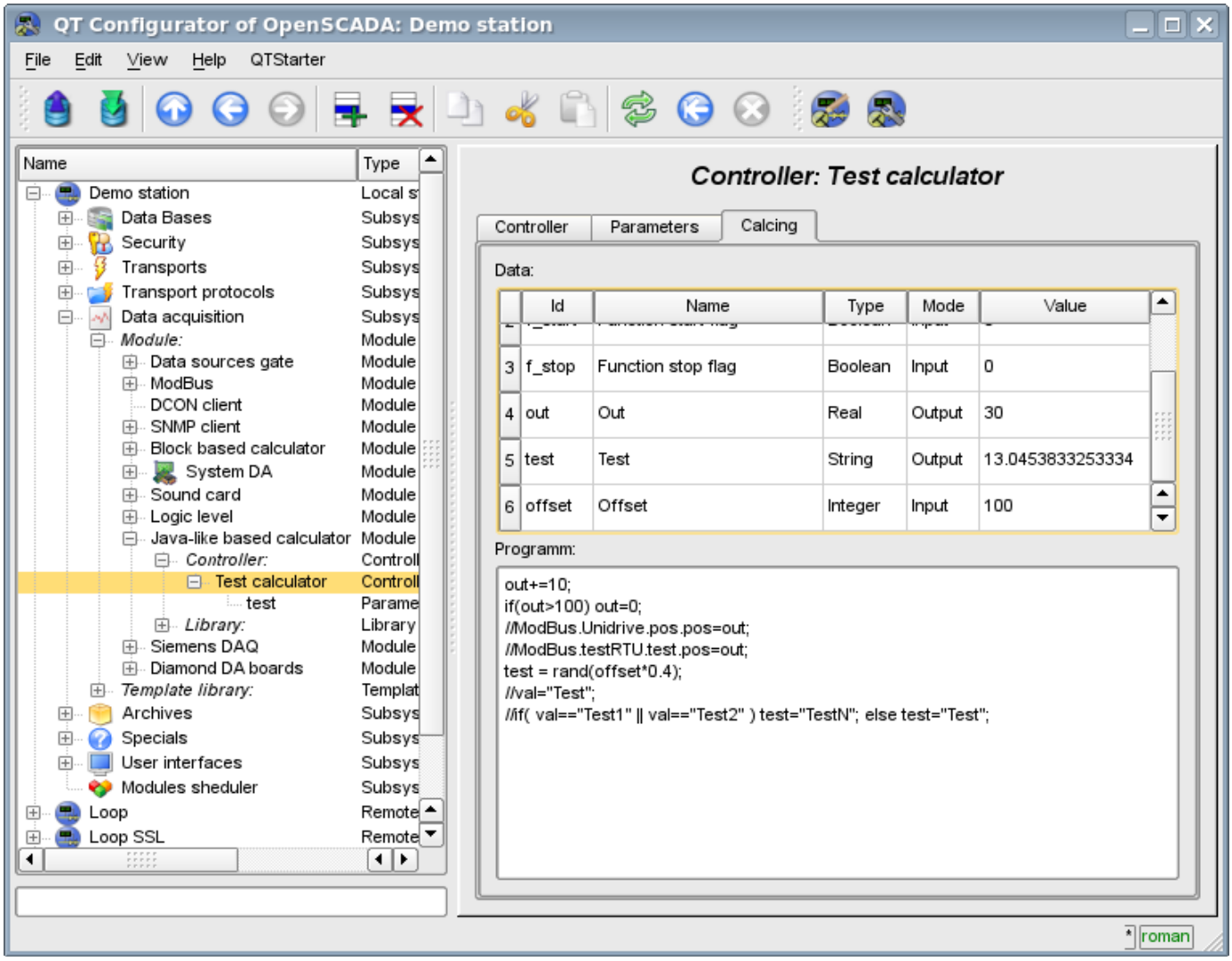

Fig.3. Tab “Calculations” of the controller.

3. The parameter of the controller and its configuration

Parameter of the controller of the module executes the function of providing the access to the results of computation of the controller to the system OpenSCADA by attributes if the parameters. Configuration tab contains only one specific field of the, set the controller only contains a field of listing the parameters of calculated function, which should be reflected.

4. Libraries of functions of module

The module provides a mechanism to create libraries of user functions on Java-like language. Example of the configuration tab of the library is depicted in Figure 4. The tab contains the basic fields: status, identifier, name and description, and also address of the table, in which the library is kept. In the “Functions” tab of the library besides the list of functions the form of copying functions is contained.

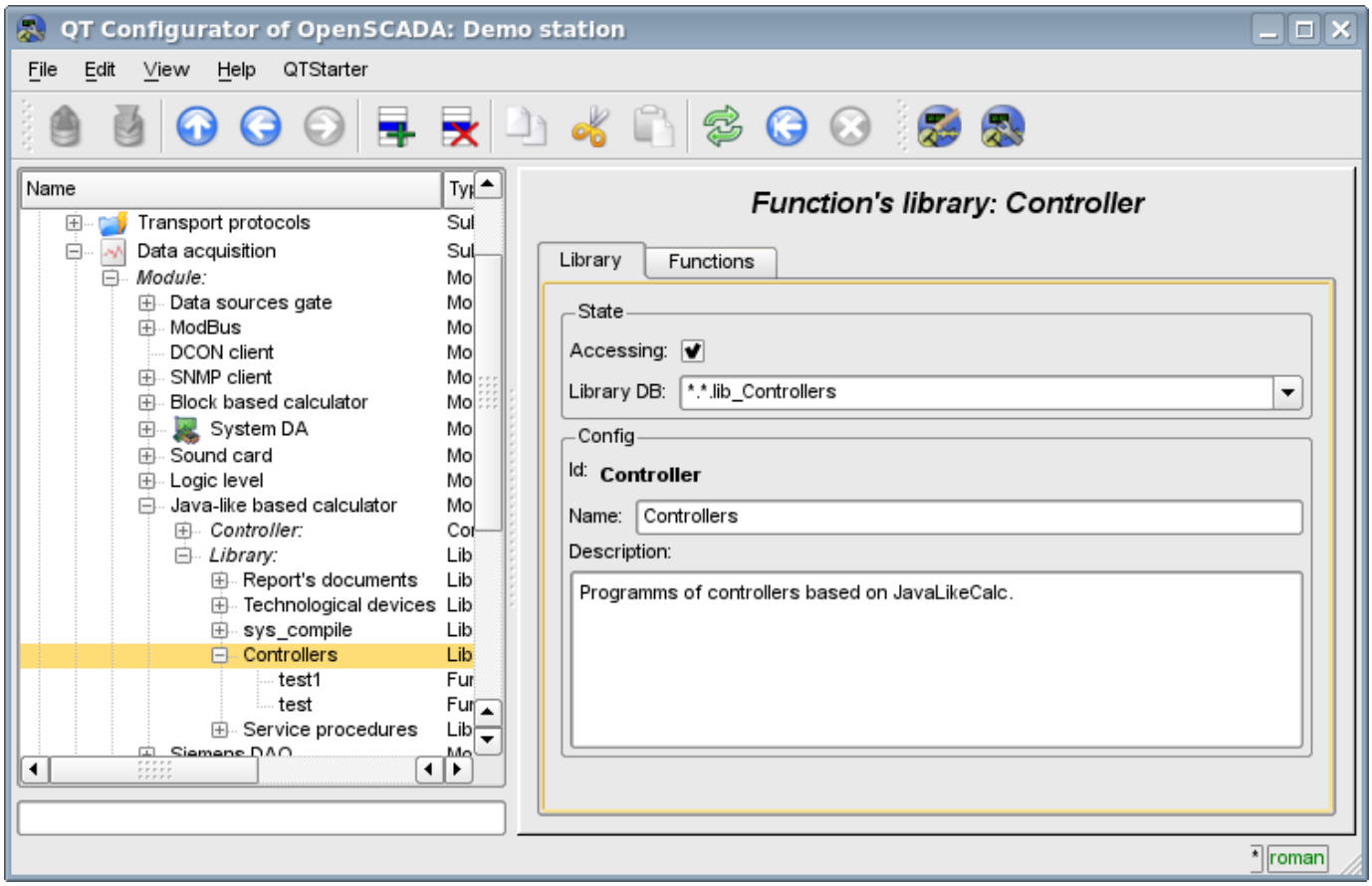

Fig.4. Tab of the configuration of the library.

5. User functions of the module

Function, as well as the library, contains the basic configuration tab, tab of the formation of the program and the parameters of function (Fig. 1), as well as the performance tab of the created function.

6. User programming API

Some objects of the module provides functions for user's programming.

The object "Functions library" (SYS.DAQ.JavaLikeCalc["lib_Lfunc"])

- *ElTp {funcID}(ElTp prm1, ...)* — call the library function *{funcID}*. Return result of the called function.

The object "User function" (SYS.DAQ.JavaLikeCalc["lib_Lfunc"]["func"])

- *ElTp call(ElTp prm1, ...)* — call the function with parameters *<prm{N}>*. Return result of the called function.