MORE COMPLEX TEST CASES

Objectives

At the end of this section, you will be able to

- Create test cases for functions whose result is not returned by the function
- Create tests for functions using “pass by address” instead of “pass by value”

testIDEA

04

Contents

MORE COMPLEX TEST CASES

1	EC-LIB© Basic knowledge	3-9
2	EC-LIB© Preparations - Set ID automatically	10-11
3	Create a base test	12-15
4	Create a derived test	16-20
5	Run test case	21
6	Summary	22-23

testIDEA

1 TESTING REAL CODE

The previous unit focused on a fictive piece of application code, simple enough to be used as an example but not very realistic.

In this unit we introduce a real function from a commercially available math library that performs floating point arithmetic using fixed point calculations. This is useful on microcontrollers that do not support floating point arithmetic natively but need to handle floating point numbers.

This example will also be used in Unit 05 again to introduce the import of test vectors into testIDEA.

We start by introducing the code and some concepts so that tests can be created.

EC-LIB© - A fixed point math library

- Aimed at microcontrollers without hardware floating point support
- Available as source code library
- “Square” function used here as a real-world example for unit testing

1 EC-LIB© BASIC KNOWLEDGE - CONCEPT

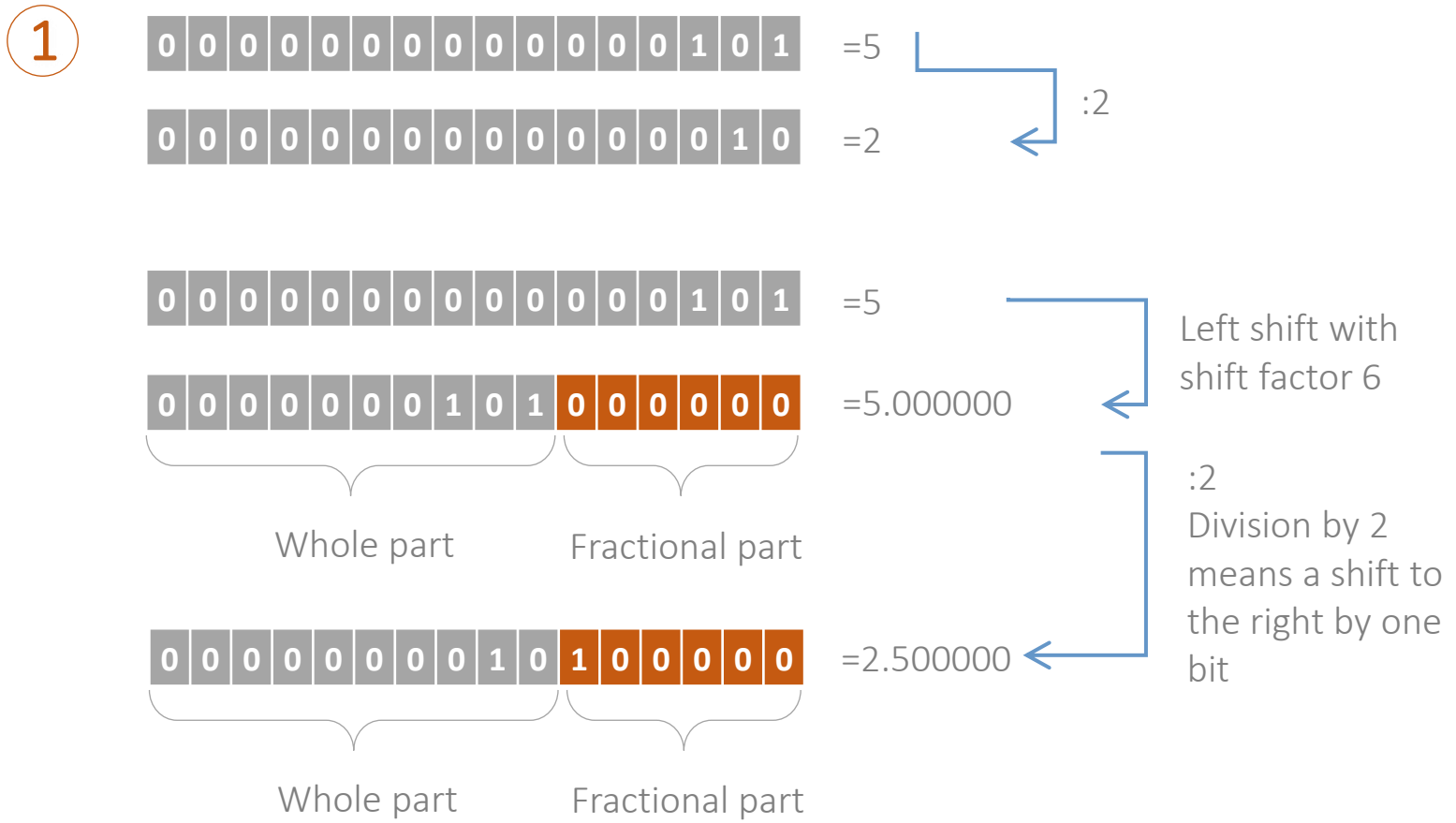
In floating point arithmetic we know that 5/2 = 2.5.

But what if we calculate with integers?

1 **Using integer arithmetic:**

5/2=2 (rounded to negative infinity)

This results in the fractional part of the result being lost.



1 EC-LIB© BASIC KNOWLEDGE - CONCEPT

In floating point arithmetic we know that 5/2 = 2.5.

But what if we calculate with integers?

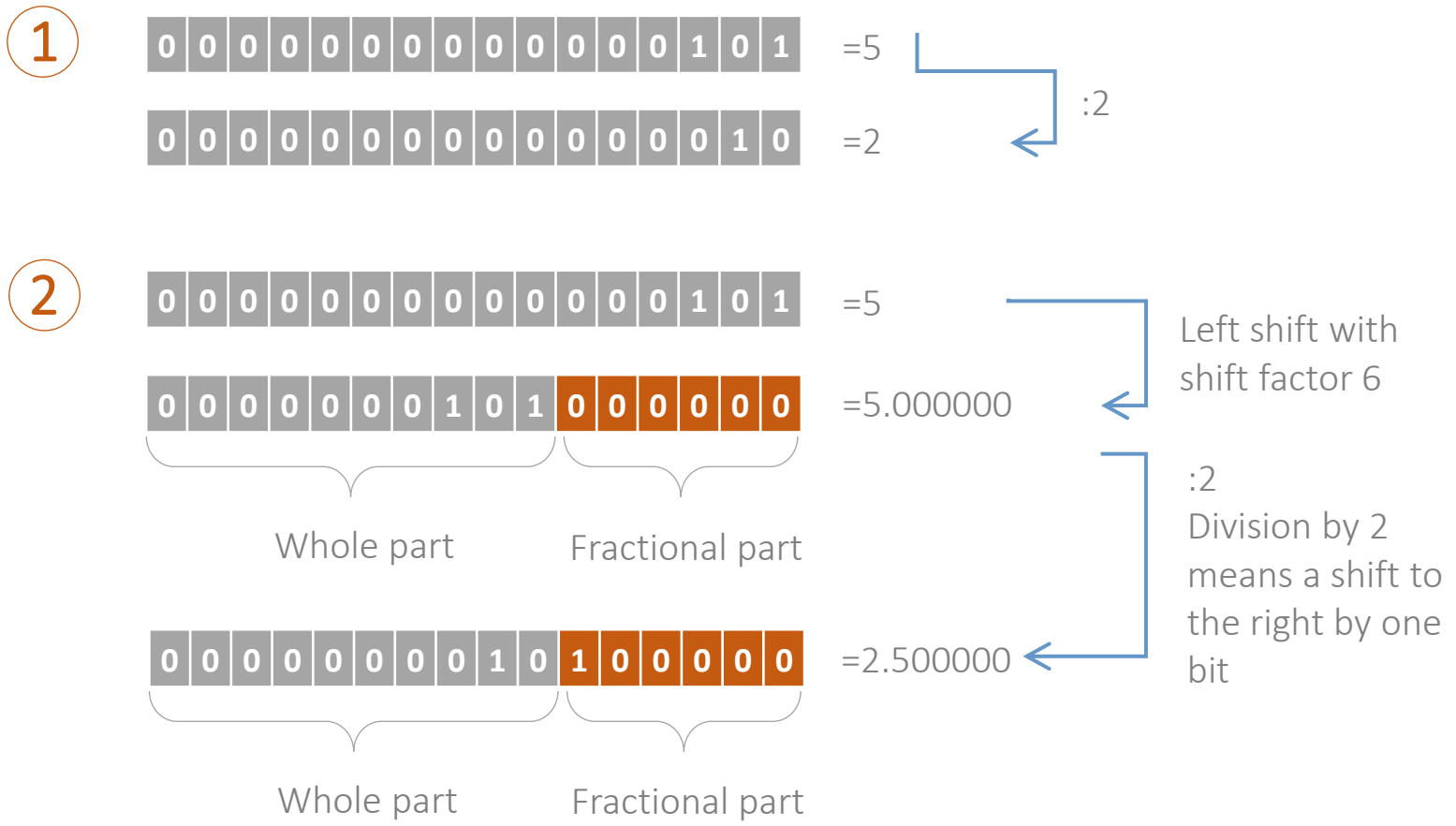
1 **Using integer arithmetic:**

5/2=2 (rounded to negative infinity)

This results in the fractional part of the result being lost.

2 **This loss of accuracy can be avoided by using shifted values:**

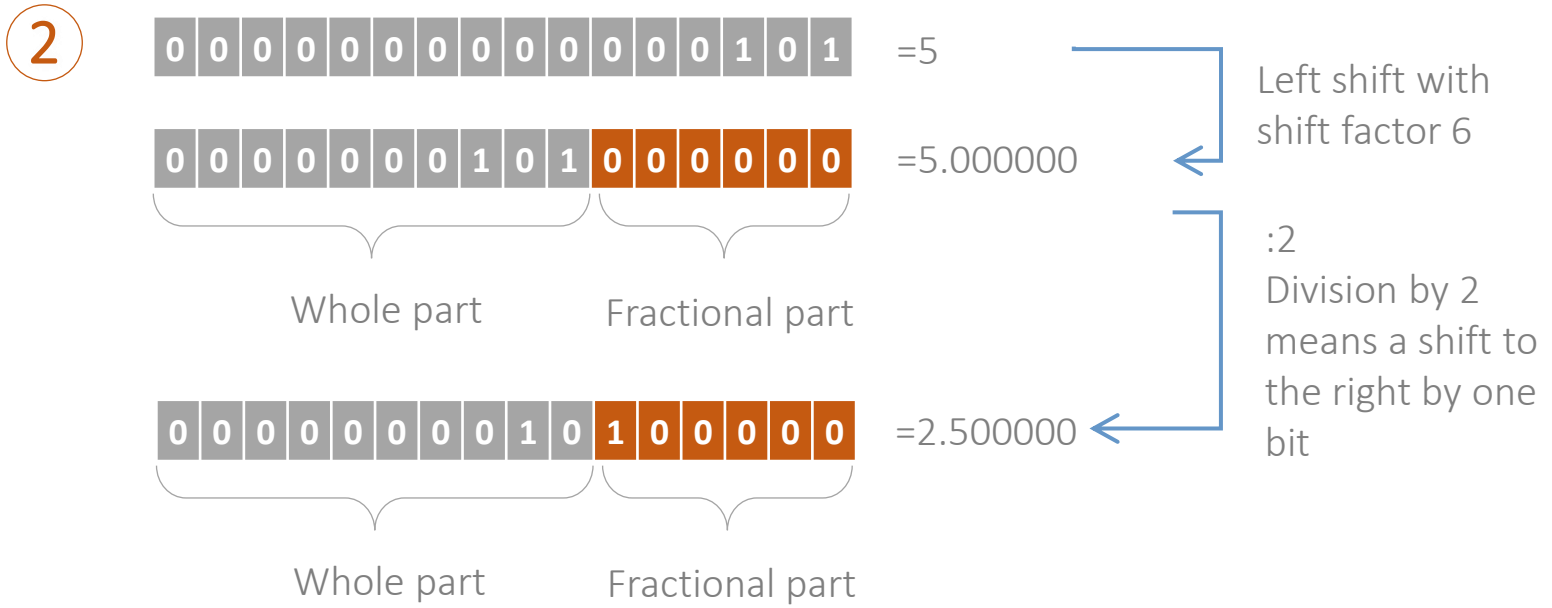
Here we divide our value range in a whole and fractional parts by shifting (in this example with a shift factor of 6).



1 EC-LIB© BASIC KNOWLEDGE - CONCEPT

The decimal place values are calculated by the formula shown, with 6 bits in the fractional part providing 2^6 = 64 possible combinations.

If the values used were to require more than 10 bits to represent them, and a 6 bit left shift were to be performed, we would loose information on the left hand side and thereby create an overflow. Consequently we have to calculate the size of the shift factor precisely and weigh it against the accuracy we will require throughout the application and calculations.



$$\frac{\text{decimal value of fractional part}}{\text{number of combinatorial possibilities for fractional part}} = \frac{32}{64} = 0.5$$

1 EC-LIB© BASIC KNOWLEDGE - CONCEPT

Generally speaking, it is possible to use this fixed point arithmetic approach not only for binary values (base 2) but with any base.

The EC-LIB© uses only base two. Thus, all calculations can be represented by shifts.

EC-LIB© uses **fixed data types**, that already provide us with the **required shift factor** which is fixed with a declaration, e.g.:

ECLIB_fix16_4sr(parameter1)

… declares a 16-bit integer called parameter1 with the related shift factor 4 (*_4sr* means a right shift with factor 4)

ECLIB_fix16_5sl (parameter2)

… declares a 16-bit integer called parameter2 with the related shift factor -5 (*_5sl* means a left shift with factor 5)

1 EC-LIB© BASIC KNOWLEDGE - CODE FOR SQUARE FUNCTION

With respect to unit testing, the function *ECLIB_Sqr_16* is considered complex for the following reasons:

- It has no return value provided by the function itself (function returns 'void'). Instead, the result is returned by address in one of the parameters passed into the function.
- Some of the parameters are passed by reference, not by value.
- The parameters, as used in the source code, are 'hidden' by the use of macros. These cannot be used within testIDEA.

```
void ECLIB_Sqr_16 (ECLIB_rcv_fix16(*res), ECLIB_rcv_fix16(par)) {
 s32 res_32;
 if (par == ECLIB_S16_NAN) {
  *res = ECLIB_S16_NAN;
 } else if (par == 0) {
  *res = 0;
 }
 else if (ECLIB_bool_IsInfinity_s16_16(par) == ECLIB_TRUE) {
  *res = ECLIB_S16_POS_INF;
 } else {
  res_32 = s32_Eclib_Square_s16(par);        // a*a
  *res = ECLIB_s16_ShiftLimitTos16_s32_16(res_32,
    ECLIB_s8_LimitTos8_s32_16(-((s32)*res_sf - (2 *
(s32)par_sf)))); // a << (sf_res - 2*sf_a)
 }
}
```

1 EC-LIB© BASIC KNOWLEDGE - TESTING STRATEGY

Now a strategy is needed to test the function. One obvious choice is to input various values (2, -2, 4, etc.) and check that the square is calculated and returned (4, 4, 16, etc.)

Due to the various shifts that can go on behind the scenes, another strategy should be developed to test the limitation imposed by working with a 16-bit container to store a fixed point number. This requires the tester to understand the principles of calculating floating point numbers using this fixed point approach.

Here we will utilize a combination of both strategies to develop our tests.

Recommended possible test strategies include:

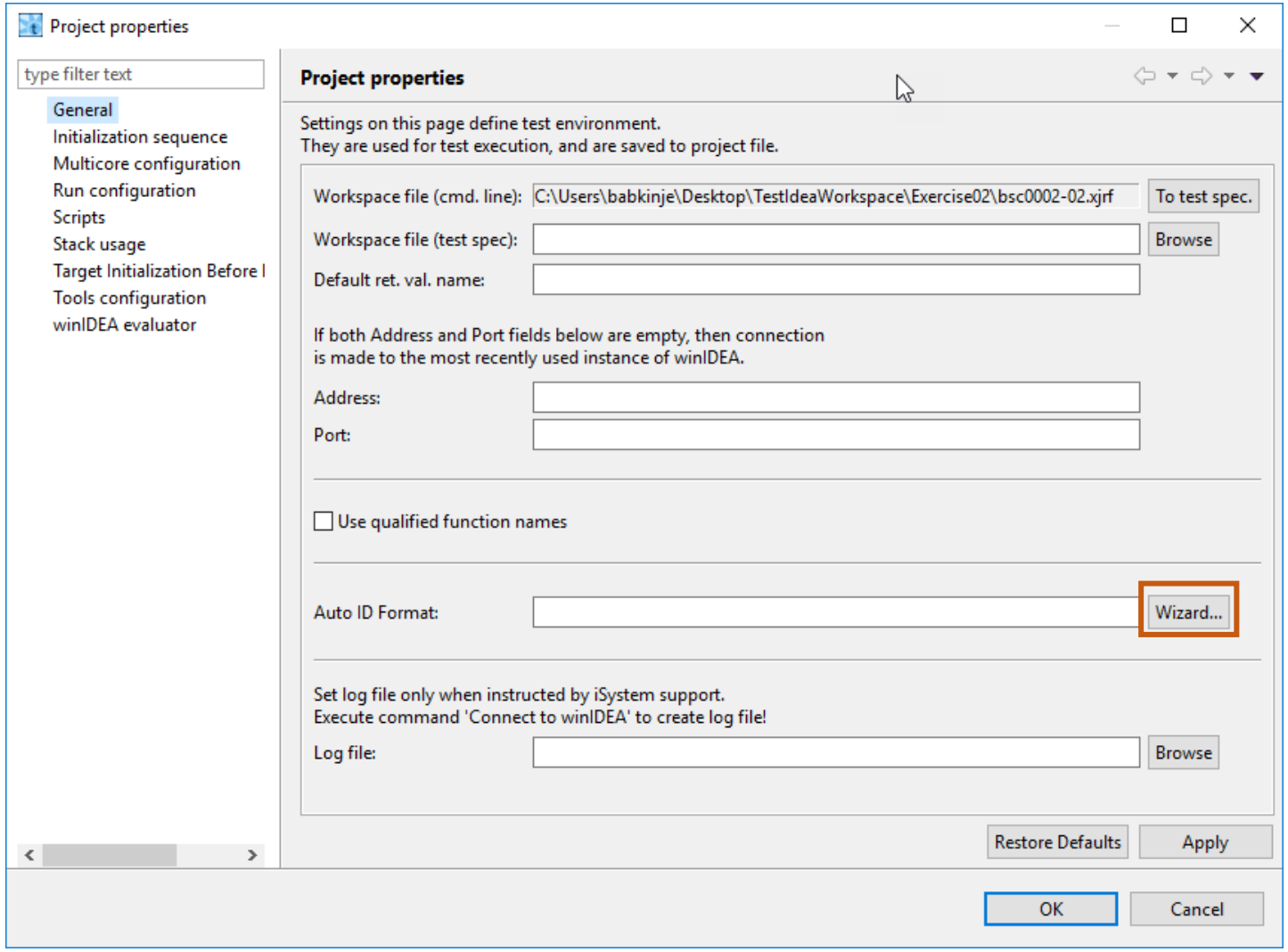
- Operational usage – since the code has a clear functionality, namely to calculate the result of (parameter)^2, simply test the math function for different input values

$$f(x) = x^2$$

- Boundary – due to the complex calculations that are hidden behind the fixed-point calculations, it makes sense to consider tests that probe the boundaries of signed 16-bit values and prove that signed-ness is maintained.

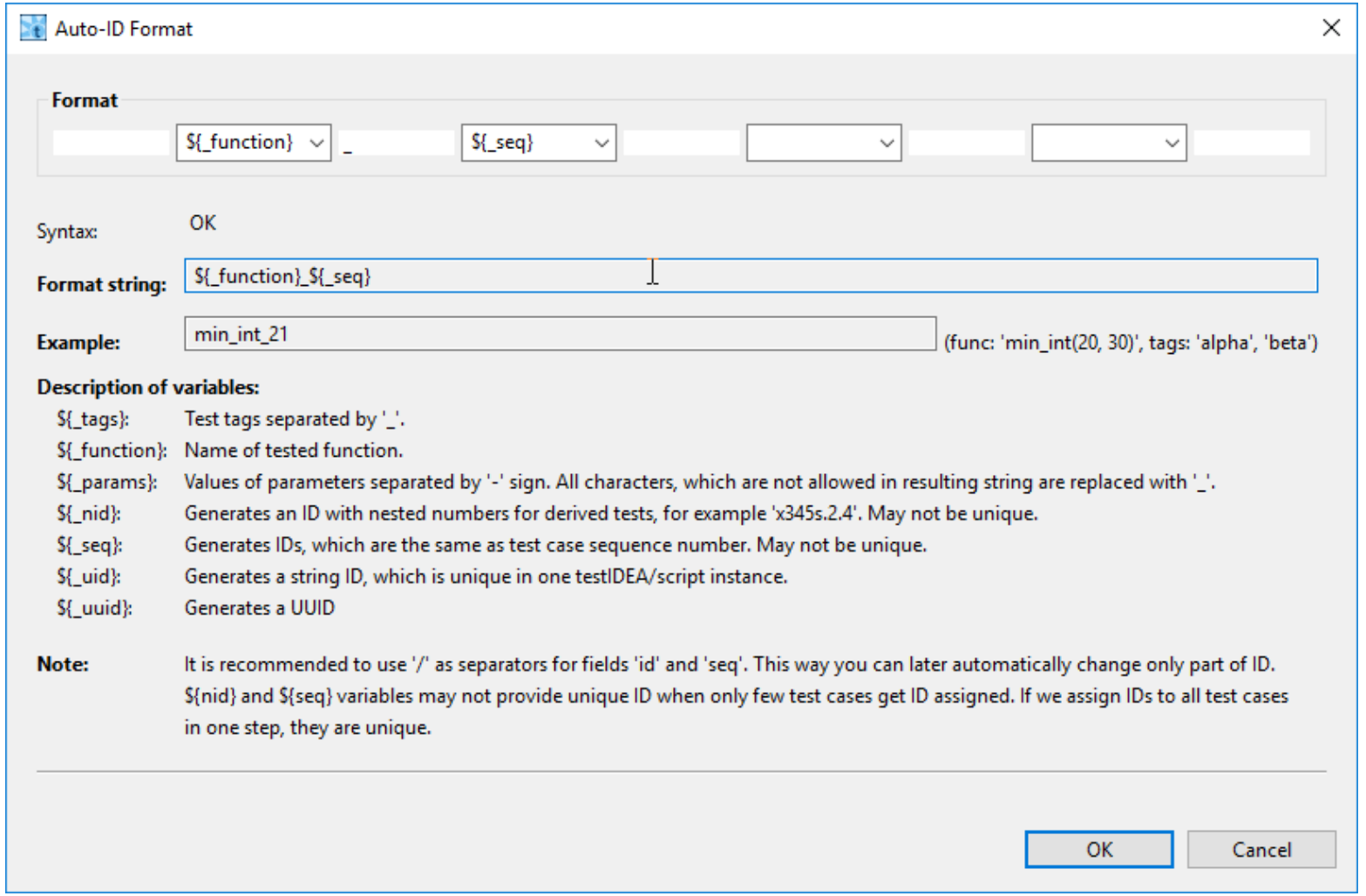
2 EC-LIB© PREPARATIONS - SET ID AUTOMATICALLY

In order to ensure that we can maintain an overview of our tests, it is recommended to turn on the automated *Test ID* generation before starting to create the tests.



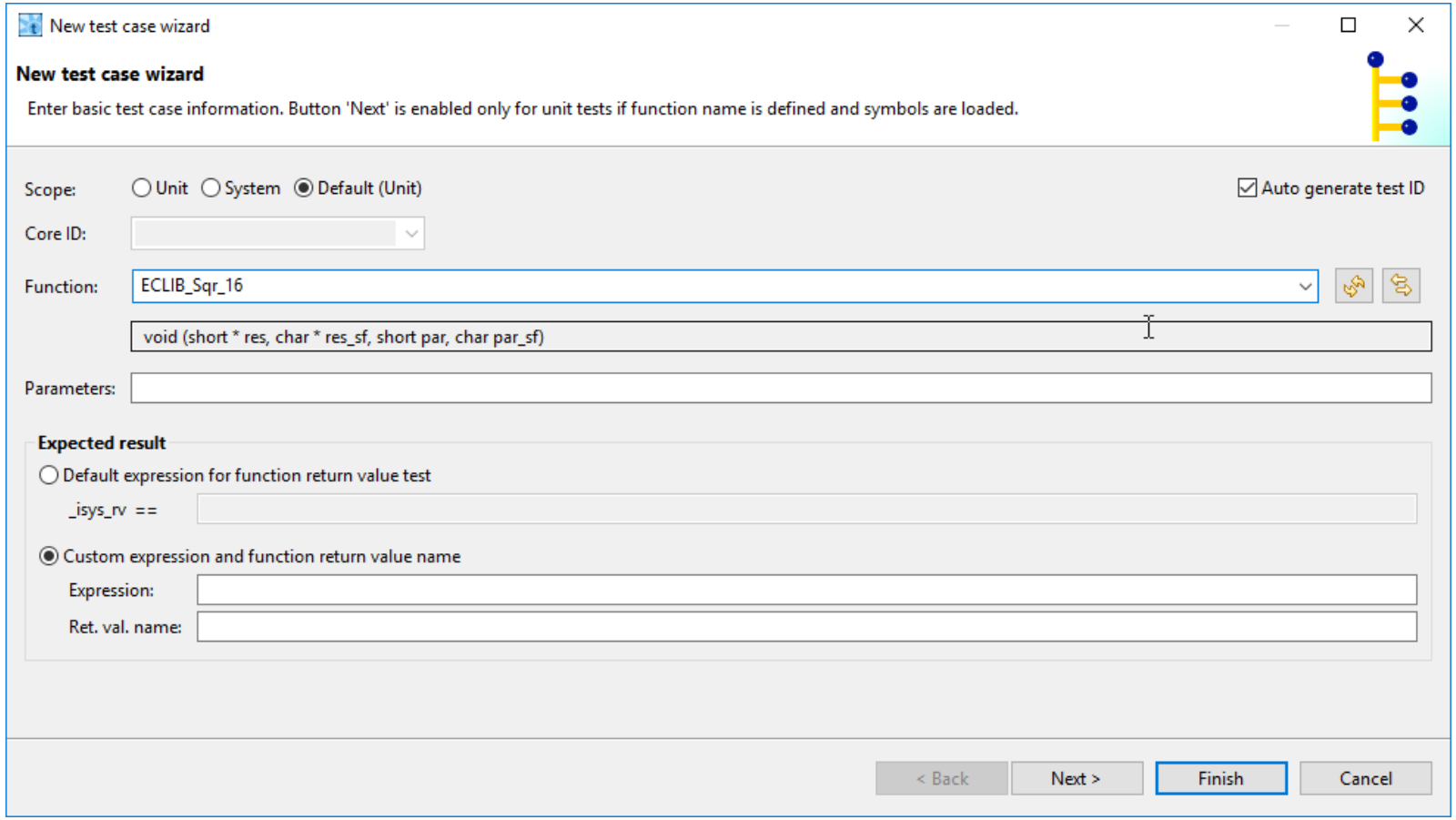
2 EC-LIB© PREPARATIONS - SET ID AUTOMATICALLY

Here we will specify that the function name and an automatically incrementing sequence number should be used for the automatically generated Test ID. An underscore "_" is used to separate these variables in the formatting string.



3 CREATE A BASE TEST (EC-LIB©)

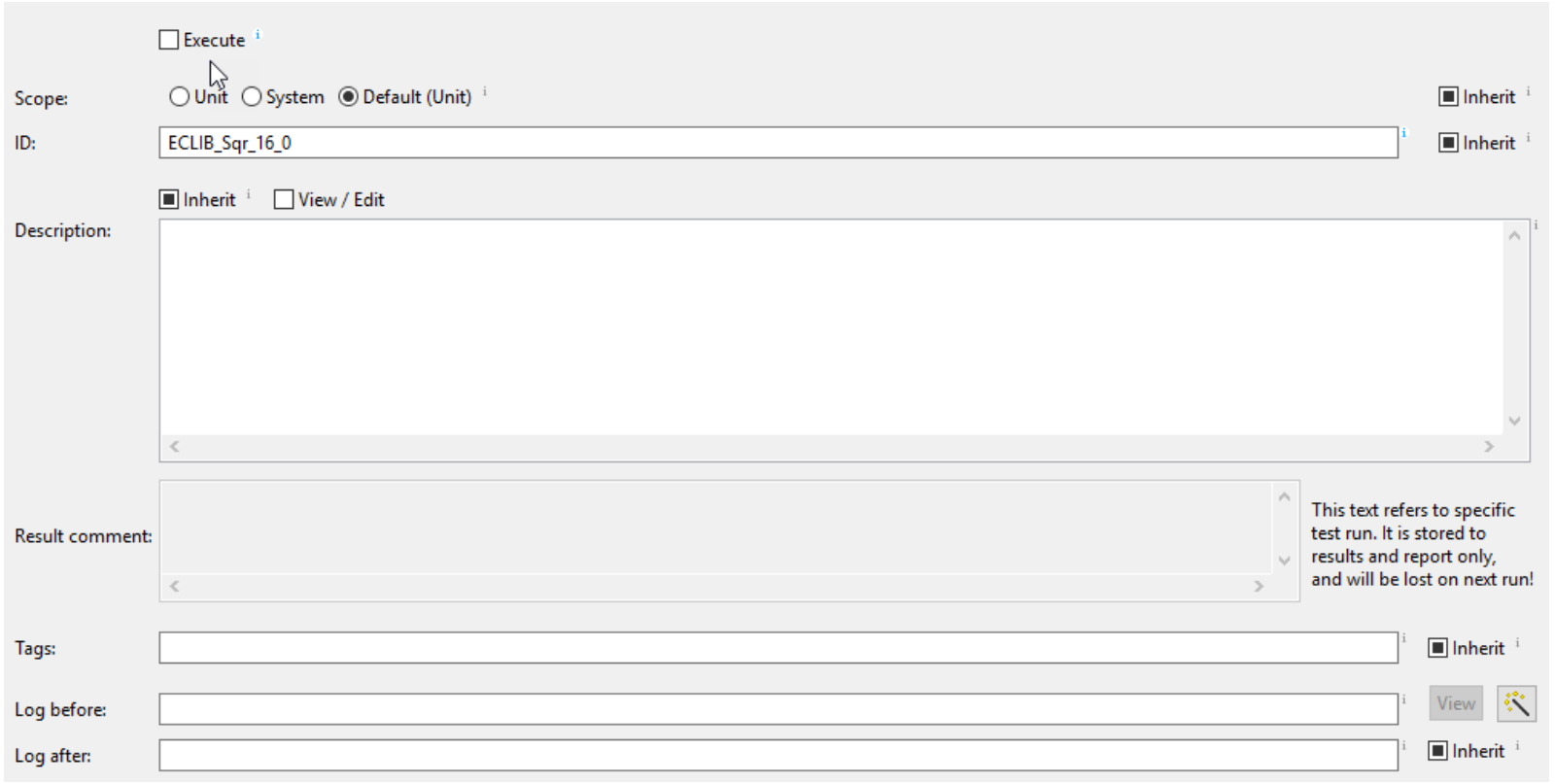
We start, as previously in Unit 03, by creating a non-executing base test. This is performed via the “New test case” option in the main menu bar. After using the *Refresh* button it is possible to select the *ECLIB_Sqr_16* function from the drop-down list.



3 CREATE A BASE TEST – META DATA (EC-LIB©)

Having created a base test we again disable the execution of this template test case.

In the meta data form, the Auto-ID setting that we defined in the steps previous will be displayed.



3 CREATE A BASE TEST – FUNCTION DATA (EC-LIB©)

In the function data form we see the chosen ECLIB_Sqr_16 function as well as the required parameters in the subline of the function field. “sf” stands for shift factor in this context and is used to define the shift factor for the input parameter and can be evaluated for the calculation result.

Output values for the result via pointers are:

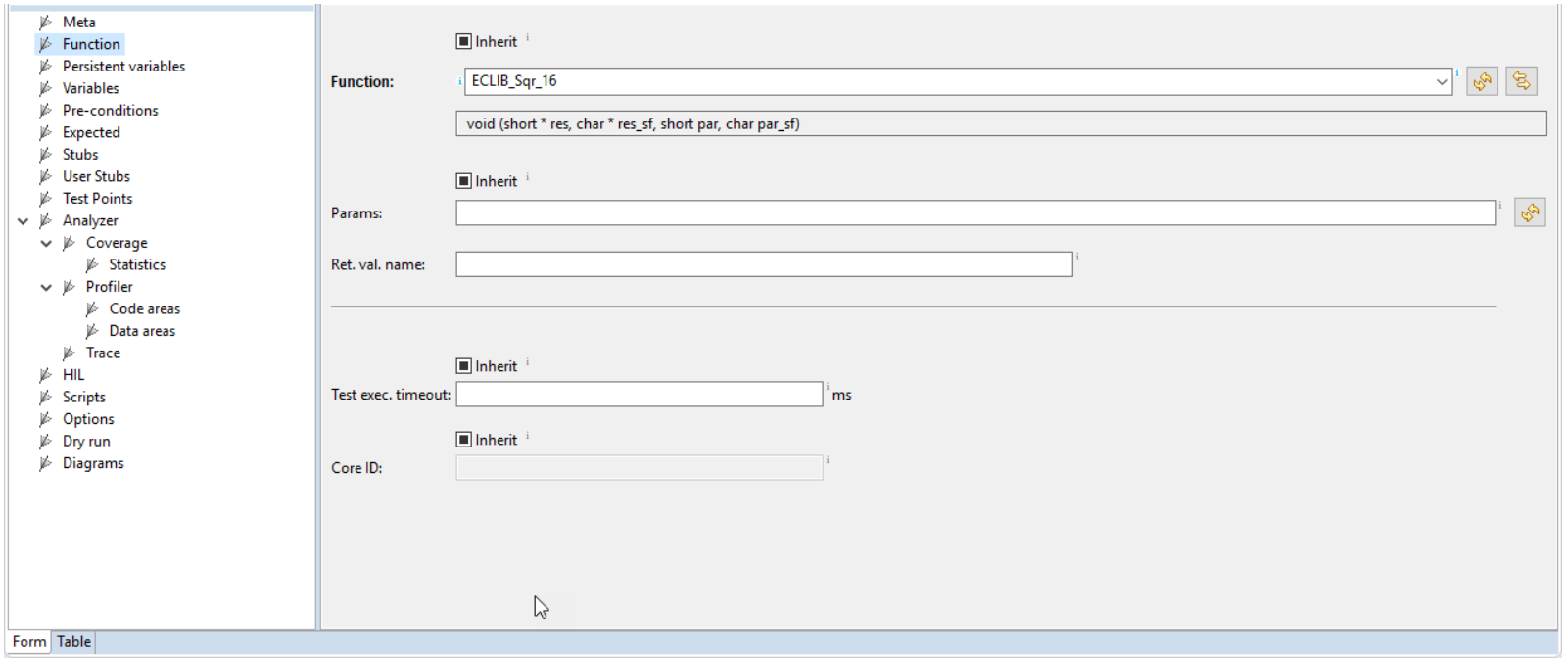
```
short*res
```

```
char*res_sf
```

Input values for the input parameter are:

```
short par
```

```
char par_sf
```



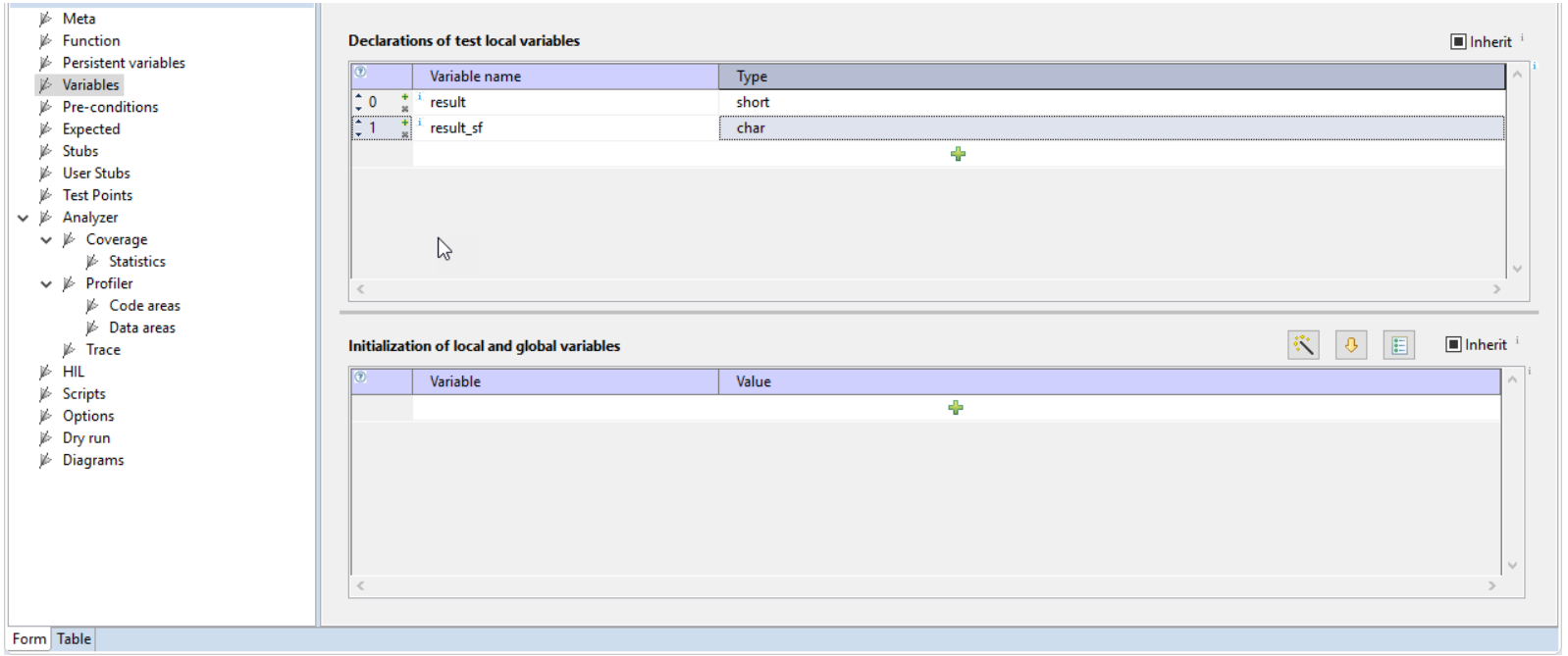
3 CREATE A BASE TEST – DECLARE VARIABLES (EC-LIB©)

Declare two input variables:

The *result* is declared as a *short* type and the *result_sf* (result shift factor) is declared as *char*.

This declaration is undertaken in the base test as the definition of these local variables will be reused in all subsequent tests. They will be created and instantiated for each test individually.

They could also be defined as *Persistent variables* but, since their content and declaration is unique to each individual test, and we do not want to maintain their value across more than one test, declaration under *Variables* is more appropriate.



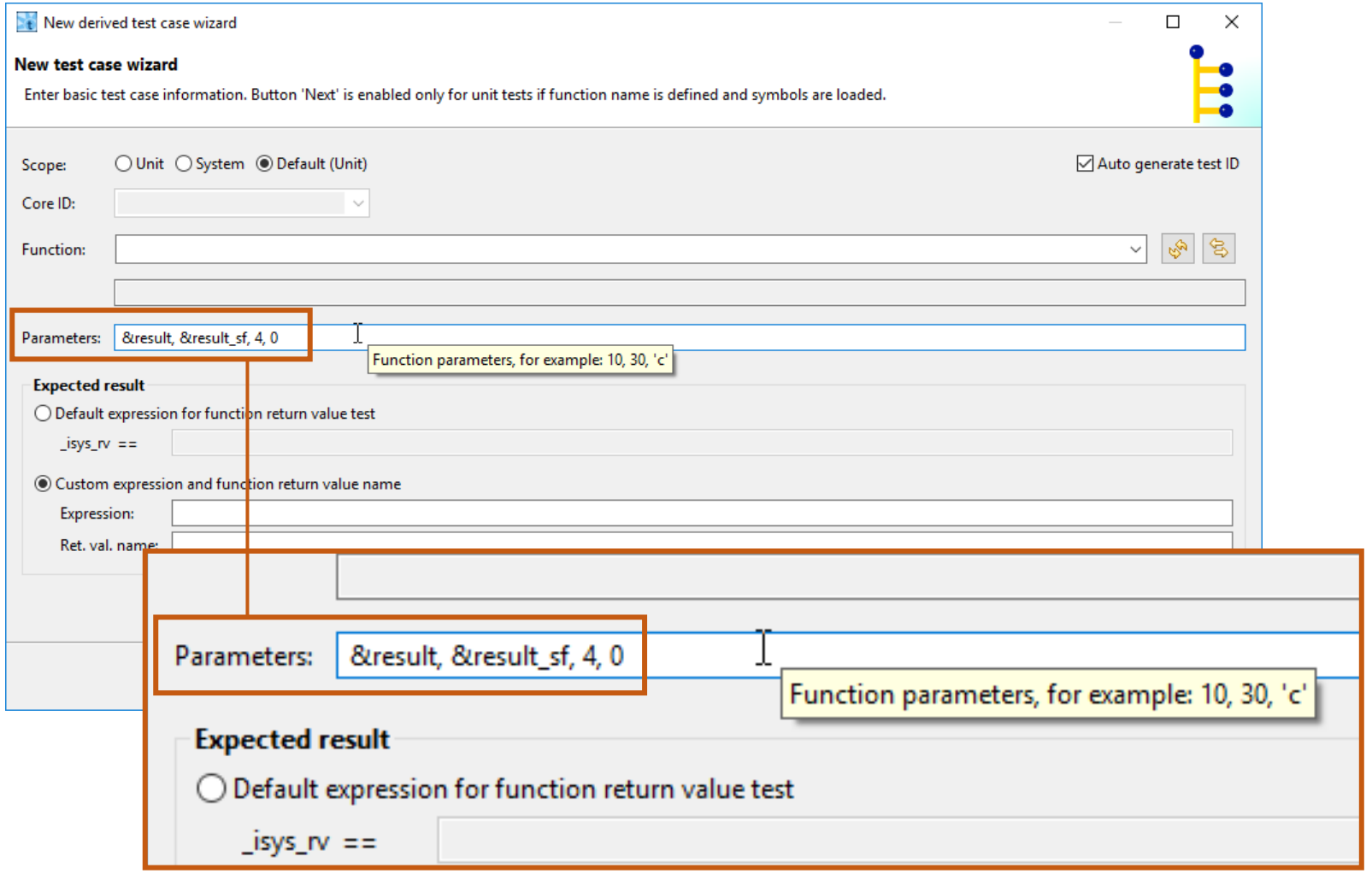
4 CREATE A DERIVED TEST (EC-LIB©)

When creating a new derived test with the test wizard the expected parameter is not visible as no function was chosen yet.

To get the parameter recommendation visible again it is necessary to select the ECLIB_Sqr_16 as the function to be tested. Once we have filled in our parameters, we have to remove this function again.

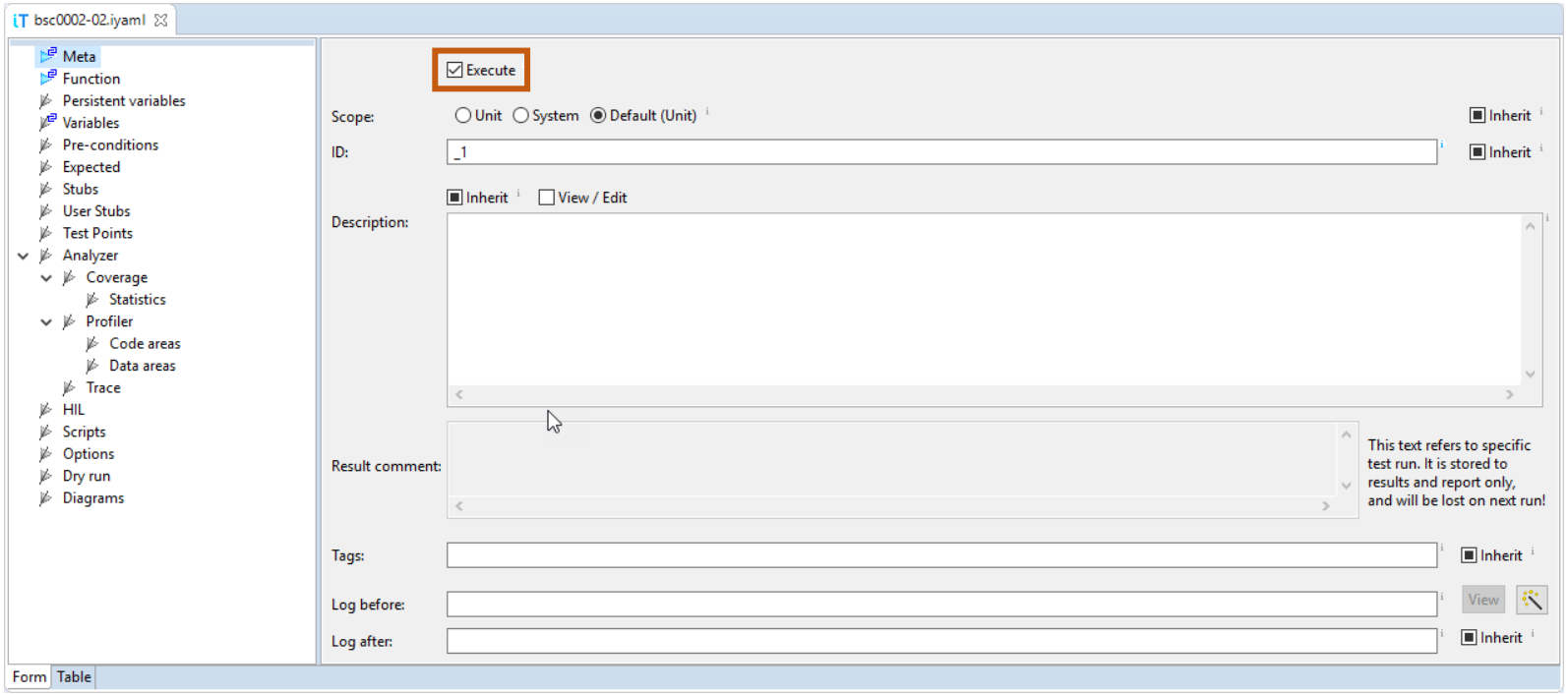
It is important to pass the addresses of *result* and *result_sf* (as is shown opposite) since they have been declared as pointers.

In this example, the input test parameter “par” is passed by value as 4, as is the shift factor “par_sf”, set to 0. Thus we will test the function for 4^2 (expect *result = 16, sf = 0*).



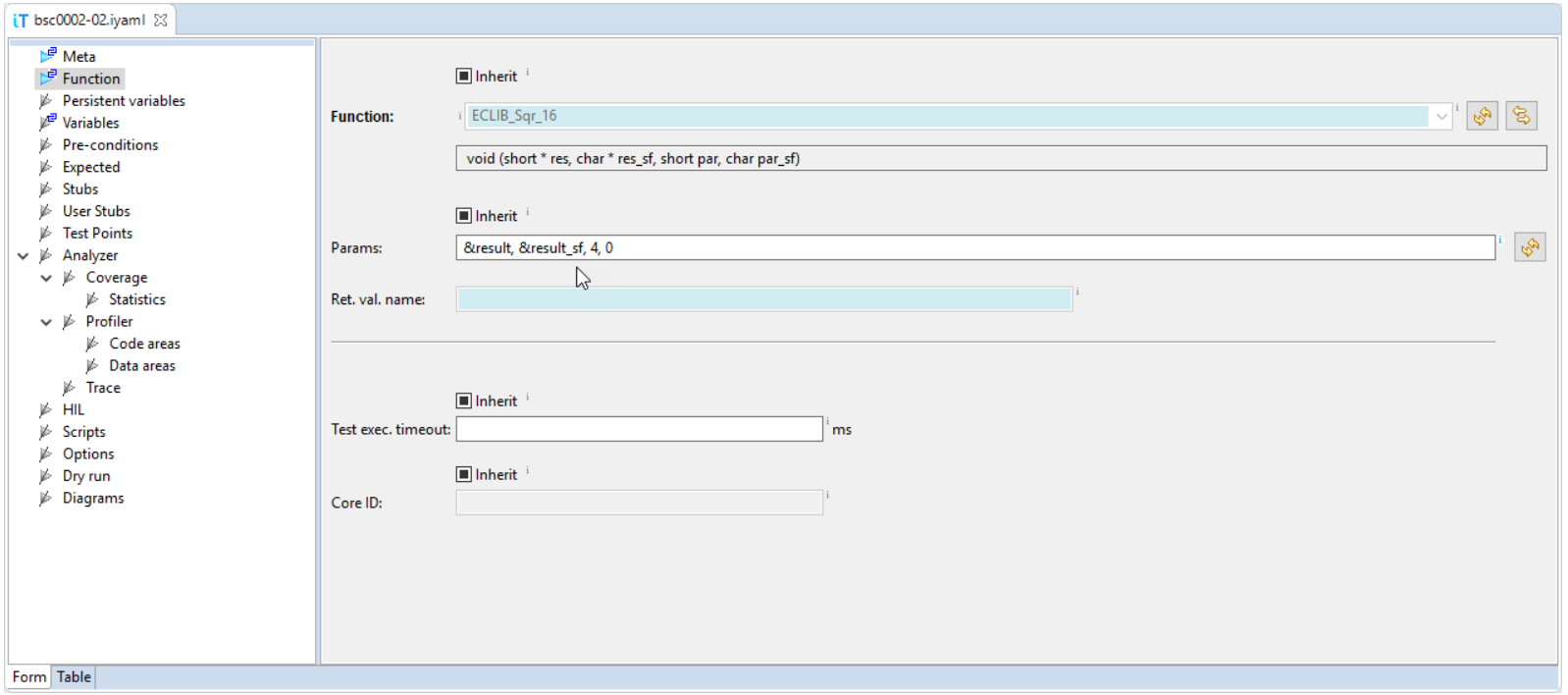
4 CREATE A DERIVED TEST – META DATA (EC-LIB©)

The meta data in the derived tests will be kept unchanged. The checkmark for execution of the test vector has to be set as this is an individual test vector with parameters and expected values and not a “template“ as the base test before.



4 CREATE A DERIVED TEST - PARAMETER (EC-LIB©)

Taking a quick look at the form view of the function data we see the inherited function name and the parameters we entered in the test case wizard before.



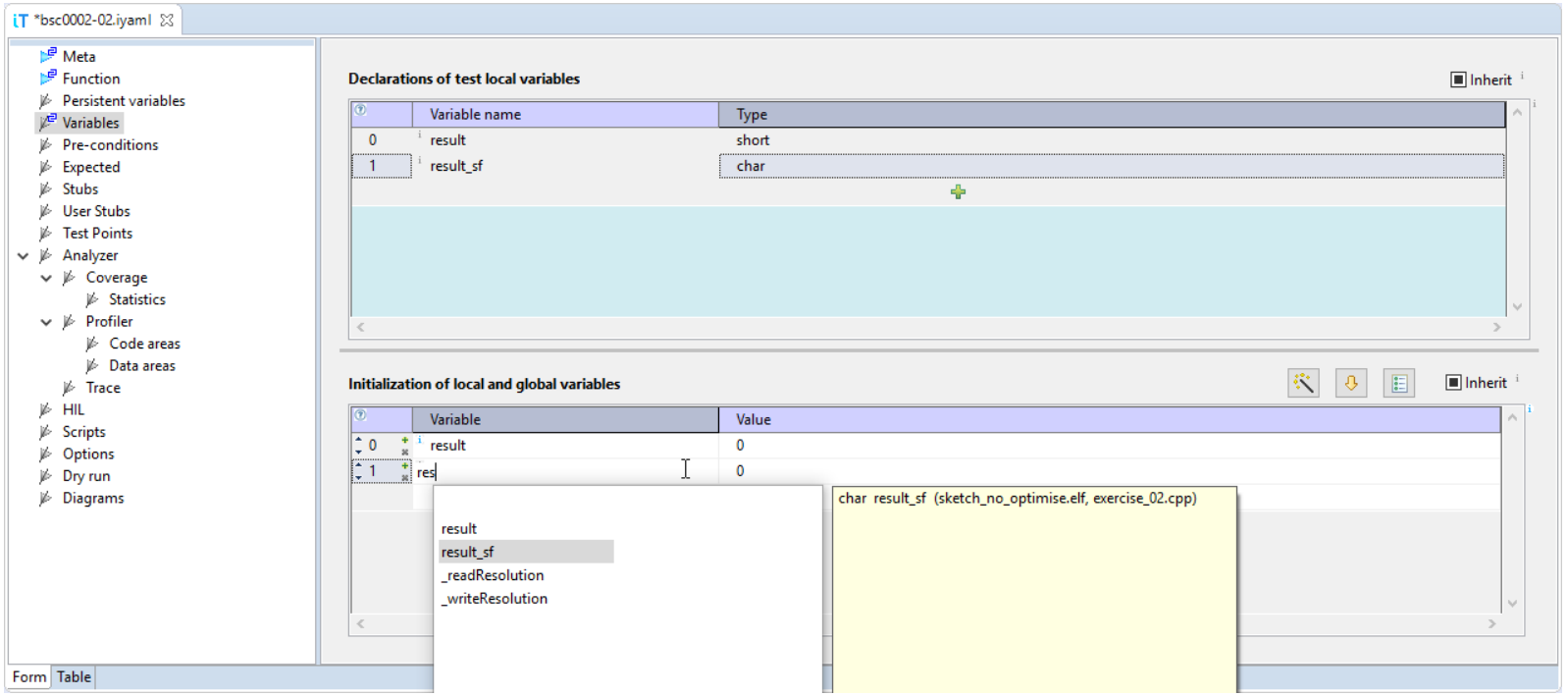
4 CREATE A DERIVED TEST – INITIALIZE VARIABLES (EC-LIB©)

Initialization of the *Variables*:

Initialization of the *result* and *result_sf* variables is not mandatory as they will be written before they are used, but it is always better to initialize each and every variable in use to a known value.

In the case that these variables are expected to contain 0 at the end of the test, other values can be used for initialization, thereby proving whether the function changed their value or not.

The shift factor for the result – *result_sf* is set to 0, meaning no shifting, which makes the first test easier to understand.

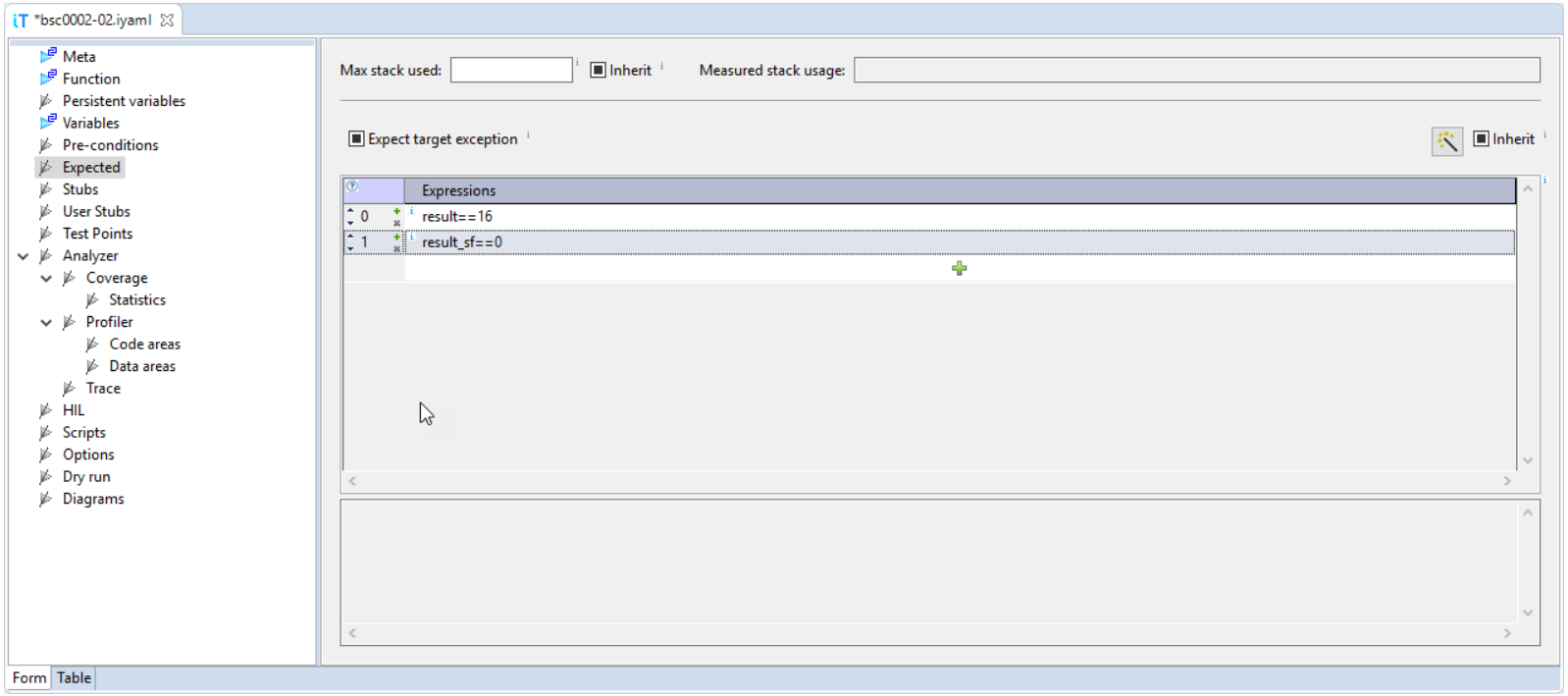


4 CREATE A DERIVED TEST - EXPECTED VALUE DATA (EC-LIB©)

In the *Expected* form view, we now enter the results we expect to have if the test passes, as we did before. The difference here is that we are expecting our results in variables defined for the purposes of the test rather than as a return value from the function, as was the case previously.

In order for the test to be considered to have passed, the variable *result_sf* should retain its value of 0.

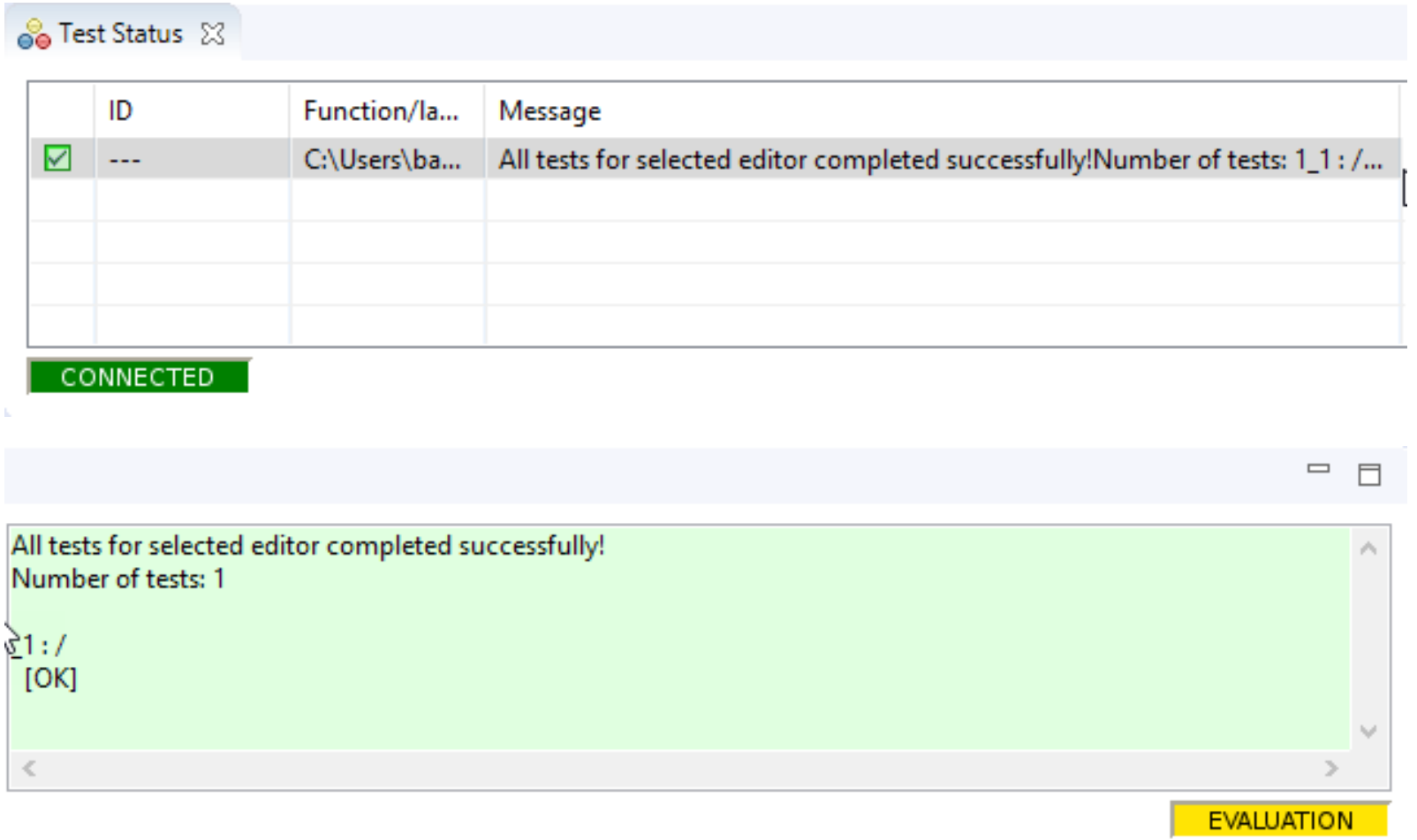
The value in the variable *result* is calculated by the square function and should be 16 (4^2).



5 RUN TEST CASE (EC-LIB©)

Having finished creating our first derived test, we can now execute the test to see if it passes or fails.

As we can see here, the function *ECLIB_Sqr_16()* passed the test calculating 4^2.



SUMMARY

testIDEA

6 SUMMARY

- Even for testing complex code we retain the same approach as before, building derived tests upon a base test.
- When passing values into a function by address, it is important to define and initialize variables for this specific purpose in the tests.
- Such variables are explicitly newly created and initialized prior to executing each test.