

# FIRST STEPS AND TEST CASES

---

## Objectives

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

- Describe the function of the key interfaces within testIDEA
- Create a base test followed by further tests built upon the base test
- Execute tests on the chosen microcontroller target
- Create, extend and modify tests using the “table” view

# testIDEA

## Contents

# FIRST STEPS AND TEST CASES

1	How the system works	3-4
2	Requirements	5
3	Connection to winIDEA	6
4	First steps with testIDEA	7-15
5	Setup the testing environment	16-23
6	Create a new base test	24-34
7	Create a derived test	35-47
8	Adding more tests - table of test cases	48-55
9	Handling test cases	56-66
10	Summary	67-68

# testIDEA

# 1 HOW THE SYSTEM WORKS



In this training package, we will mostly assume that the user knows how to approach testing their application to fulfil the demands of their application's requirement. However, even if you don't, you will pick up some tips and ideas as we show how certain types of code constructs in C and C++ can be tested.

Effective testing can only be achieved with a clear test specification. This can simply be a Word or Excel document if no other formal system is in place.

testIDEA is used to actually write the tests, providing input parameters and expected outcomes for each function or method to be tested.

Configuration  
covered in  
BSC0001

Test specification  
/ External test  
cases



testIDEA



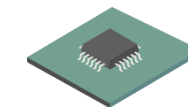
winIDEA



BlueBox™



MCU



# 1 HOW THE SYSTEM WORKS



It is important to note that testIDEA cannot automatically create tests. It does, however, through various means such as importing test vectors and automation features, help in the creation of tests.

In order for testIDEA to work, it needs access to an instantiation of winIDEA that has a working and correctly configured workspace, set up to work with the selected BlueBox™ hardware and the chosen microcontroller. The steps required to achieve this are covered in our training course [BSC0001 – Introduction to winIDEA](#).

Configuration covered in BSC0001



Test specification  
/ External test  
cases



testIDEA



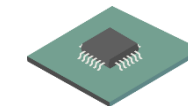
winIDEA



BlueBox™



MCU



# 2 REQUIREMENTS



Before any tests can be executed, target initialization must also occur i.e. the start-up code prior to **main()** must have been executed. This will ensure that the stack is allocated. Depending on the target microcontroller, further code may also need to be executed to ensure

- Memory spaces are configured
- MCU peripherals are initialized (clocks, ports... )
- Global data is initialized

Additionally, all functions or methods to be tested must exist in memory. Most compilers will optimize away code that is not used. You may need to write a simple application example that specifically calls all the code to be tested.

## Requirements

- A **winIDEA workspace** and suitable **configuration** for target development board
- Target **initialization** must work
- Download **executable code** onto target MCU
- All **functions** or **methods to be tested** must exist in memory

## Before starting with testIDEA

1. Download application code to target memory
2. Execute run until function *main()*

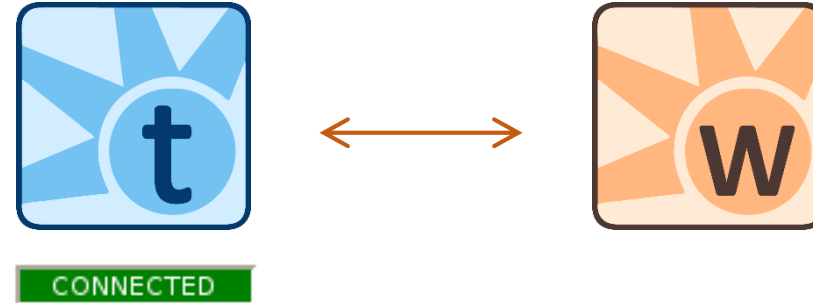
# 3 CONNECTION TO winIDEA



When creating a test specification within testIDEA, function, method and variable names must be specified.

winIDEA knows the names of all functions and variables from the debug information provided in binary files (e.g. ELF file) and testIDEA can acquire this information from winIDEA to help us with auto-completion. Therefore it is convenient to have winIDEA running when creating tests.

The connection status between testIDEA and winIDEA is shown in the bottom left corner of the iSYSTEM testIDEA window.



*winIDEA must be able to download the code in order for symbols to be available. If there is no hardware target available, switching winIDEA to demo mode is a suitable work-around.*



*If testIDEA was started from winIDEA, then it will automatically connect to this instance of winIDEA. **If we close the winIDEA instance, testIDEA will loose this connection.** In such situations you will need to close and restart testIDEA.*

# 4 FIRST STEPS WITH testIDEA



## Starting testIDEA

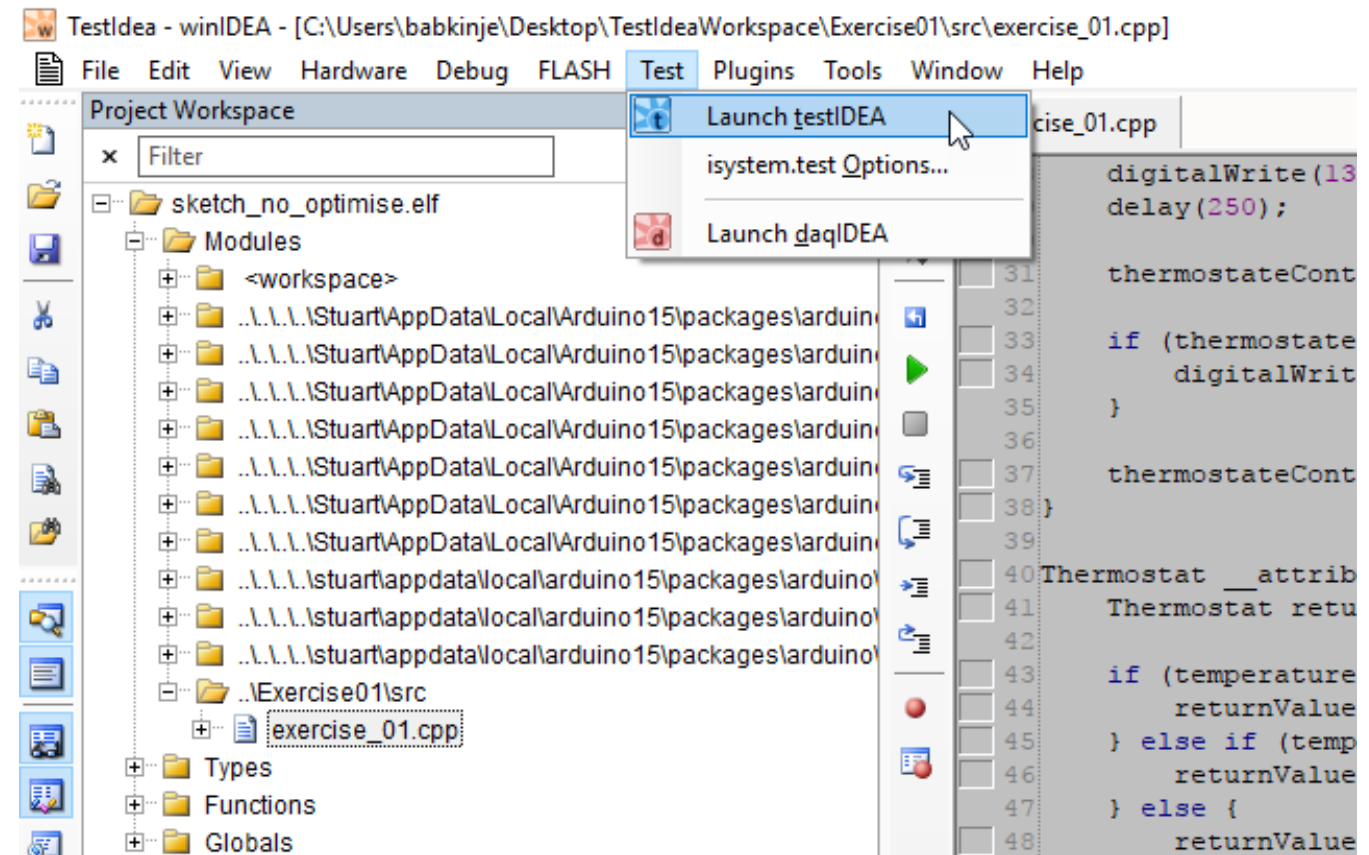
The simplest method is to use the menu option within **winIDEA**: *Test* → *Launch testIDEA* (as displayed right).

If testIDEA has already been started, the menu option from within **testIDEA iTools** → *Connect to winIDEA* enables testIDEA to connect to an existing instance of winIDEA.

Finally, simply pressing the *Refresh* button within **testIDEA** will open a dialog box asking whether we want to connect (covered in later slides).

It is also possible to configure testIDEA to connect to winIDEA automatically using *iTools* → *Preferences* → *testIDEA* → *Connect automatically*.

From within a winIDEA workspace:



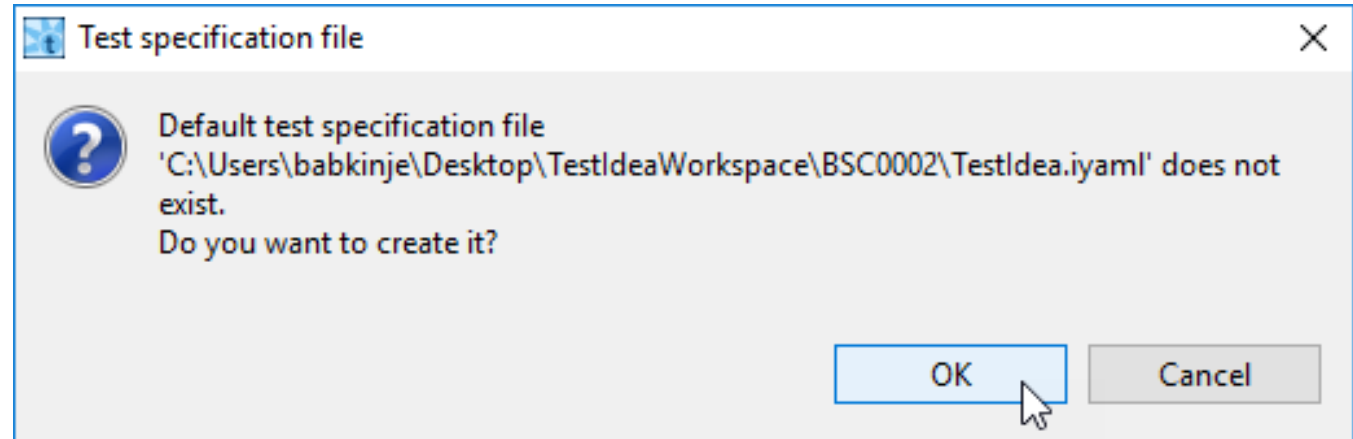
# 4 FIRST STEPS WITH testIDEA



When we first open testIDEA for a workspace for which no test specification has been created, we will be asked if we want to create an \*.iYAML file by default. It is recommended to accept this default setting as the testIDEA workspace will be stored in the correct folder (same as winIDEA workspace) automatically.



*If this window doesn't appear, it might be that there is already an existing testIDEA workspace or that it is **hidden in the background** of winIDEA window.*





# 4 FIRST STEPS WITH testIDEA

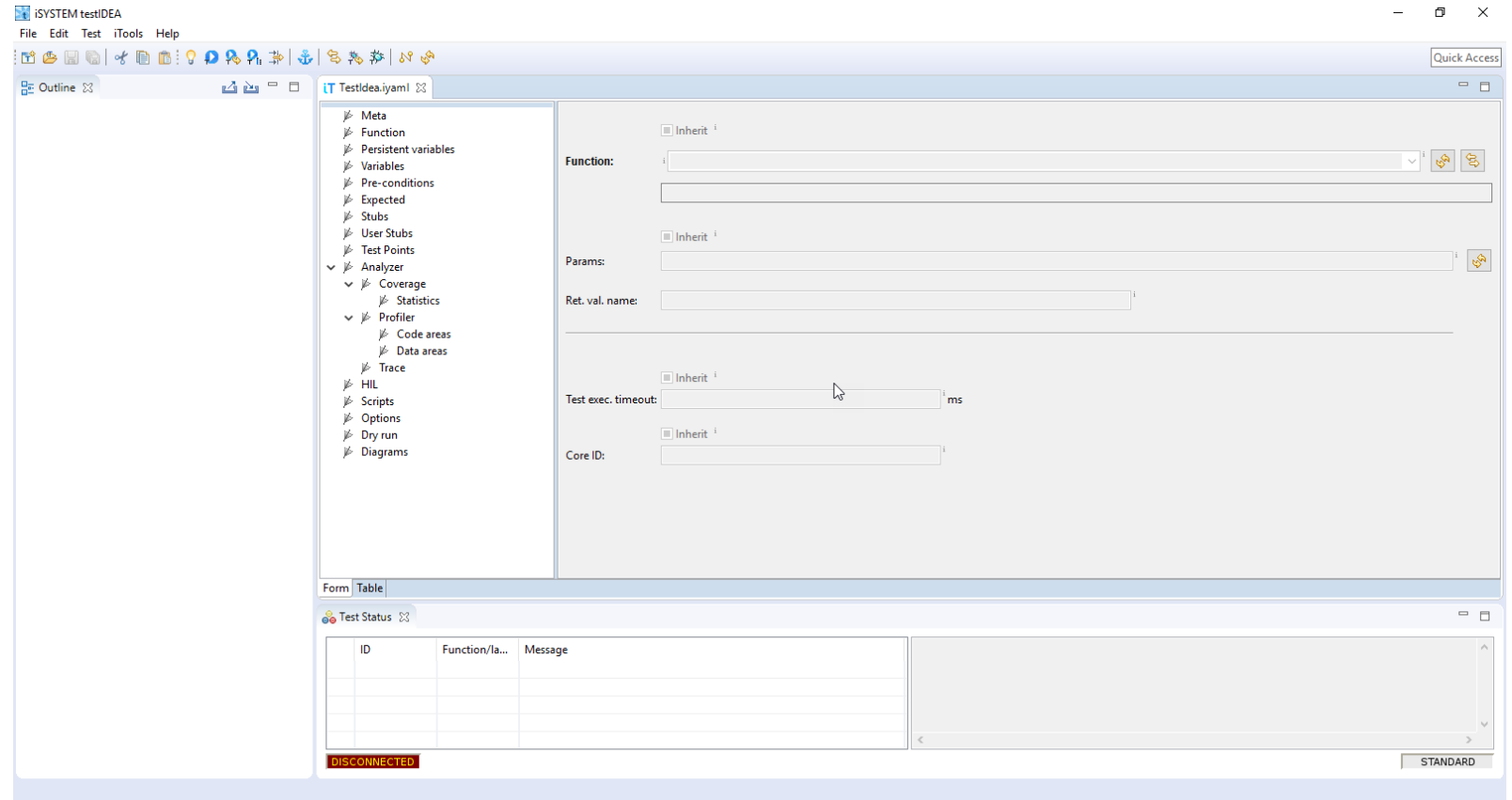


## testIDEA user interface

Once testIDEA has been started, you will be confronted by the following user interface.

The following slides guide you through the various interface areas and the key menu options.

Once this has been covered, you will be ready to start creating your first tests.



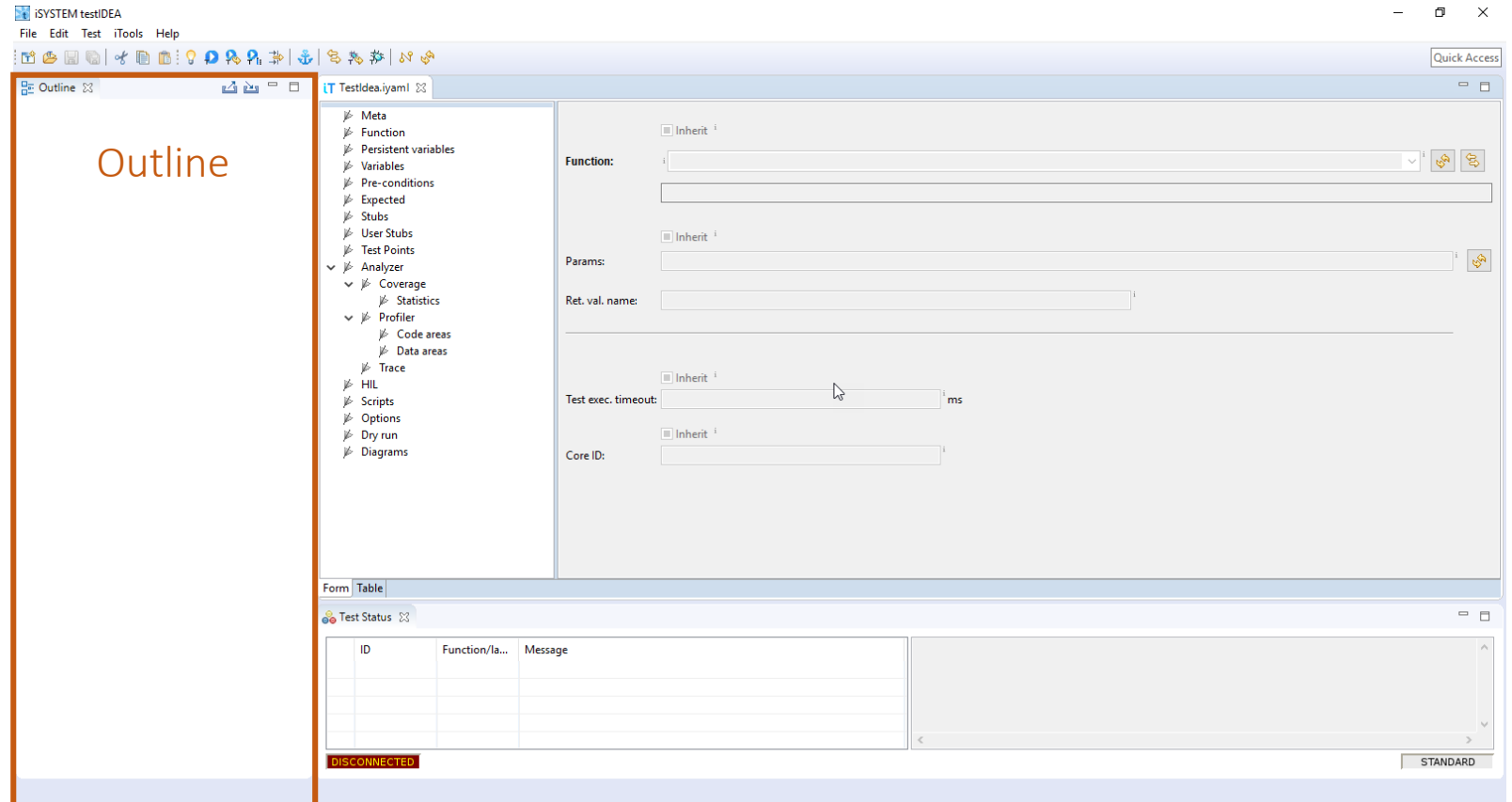
# 4 FIRST STEPS WITH testIDEA



## testIDEA user interface

### Outline

This view contains a list of all test cases. By clicking a test case in the tree, its content is displayed in Test Case Editor area. The context menu of the Outline view contains options for creating and deleting test cases.



# 4 FIRST STEPS WITH testIDEA



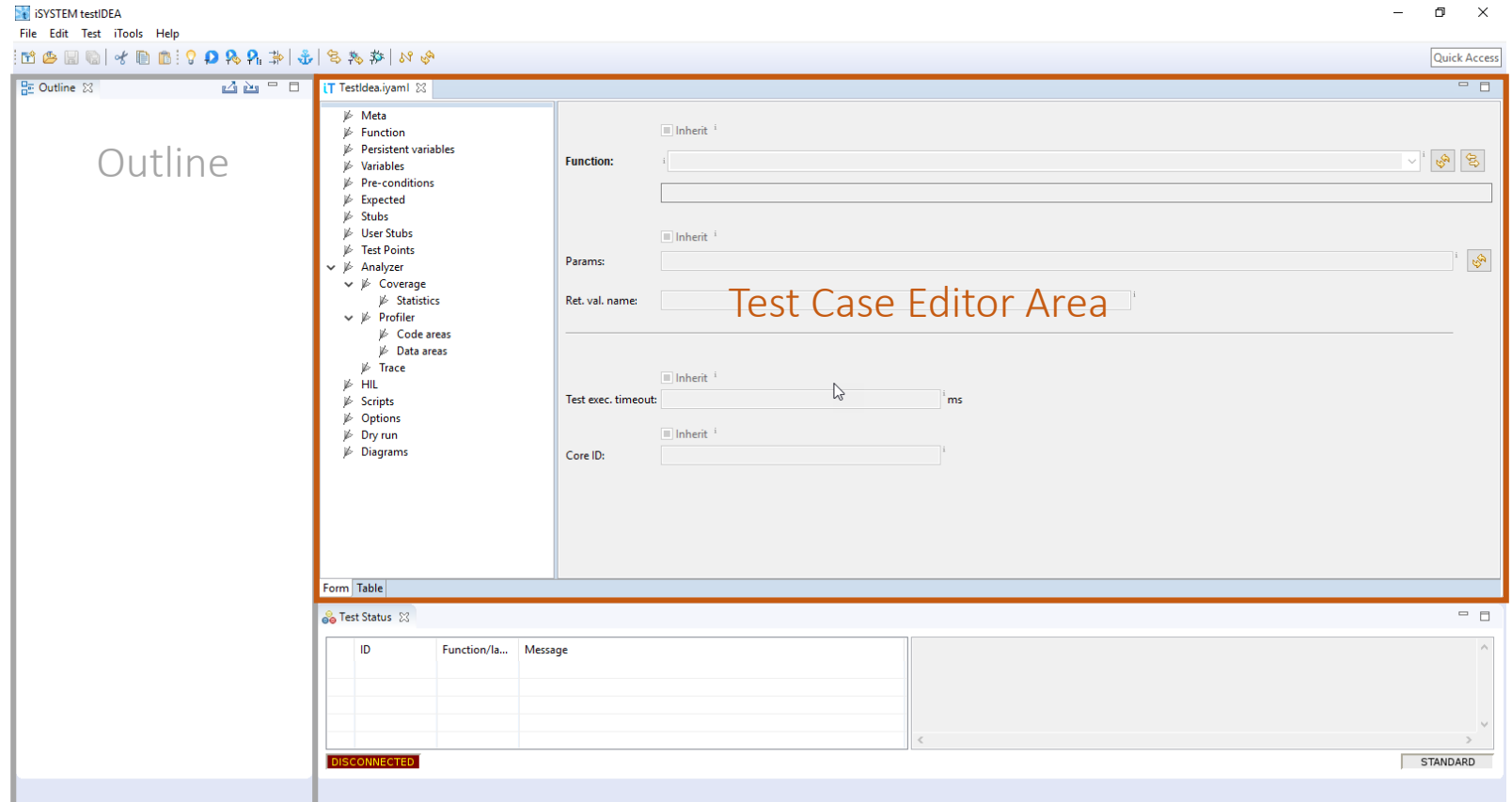
## testIDEA user interface

### Outline

This view contains list of all test cases. By clicking a test case in the tree, its content is displayed in Test Case Editor area. The context menu of the Outline view contains options for creating and deleting test cases.

### Test Case Editor Area

This area contains editors with controls for viewing and modifying a test case.



# 4 FIRST STEPS WITH testIDEA



## testIDEA user interface

### Outline

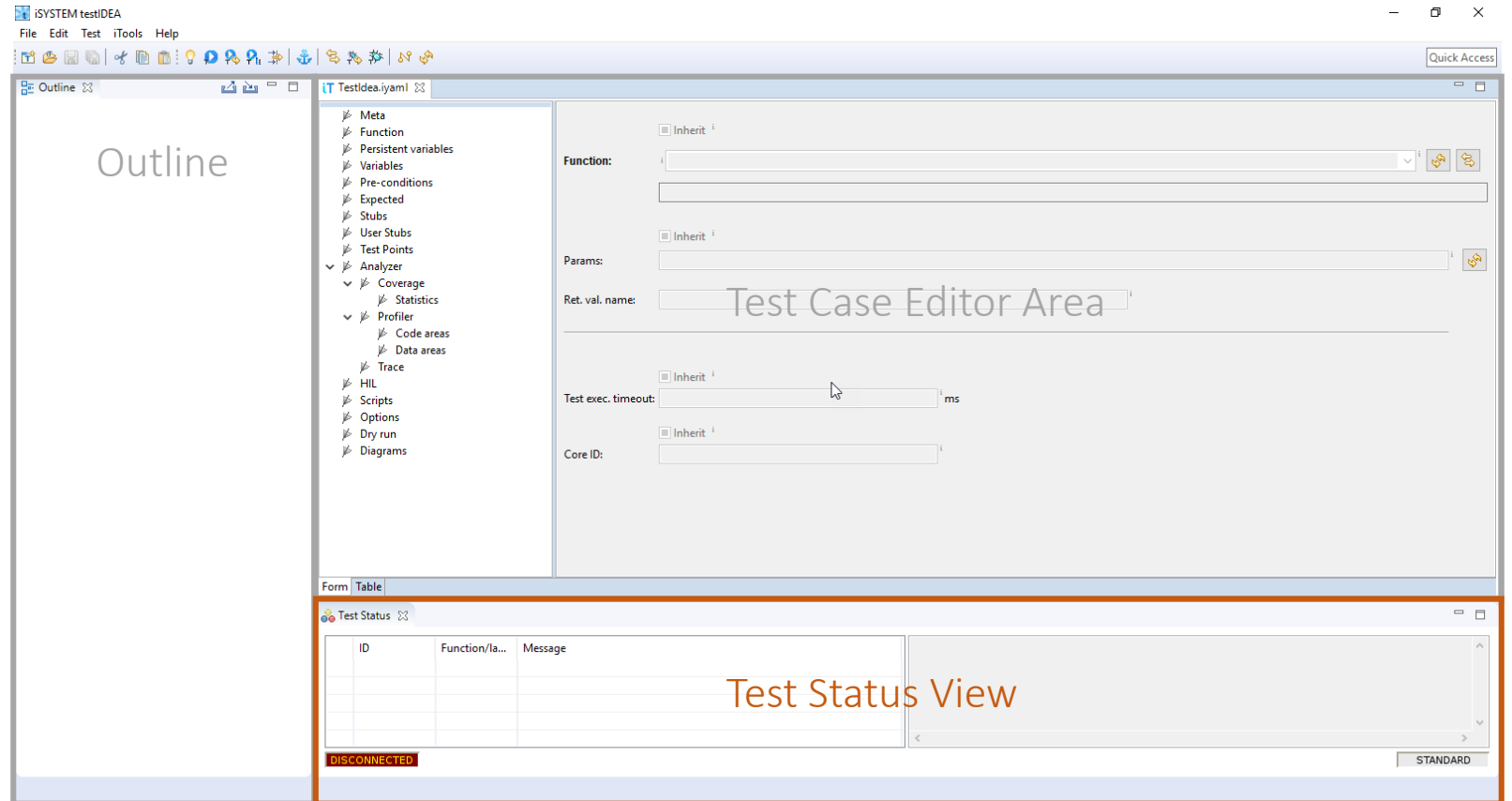
This view contains list of all test cases. By clicking a test case in the tree, its content is displayed in Test Case Editor area. The context menu of the Outline view contains options for creating and deleting test cases.

### Test Case Editor Area

This area contains editors with controls for viewing and modifying a test case.

### Test Status View

This view displays status messages. Summaries of test results and error messages during editing are displayed here.



# 4 FIRST STEPS WITH testIDEA



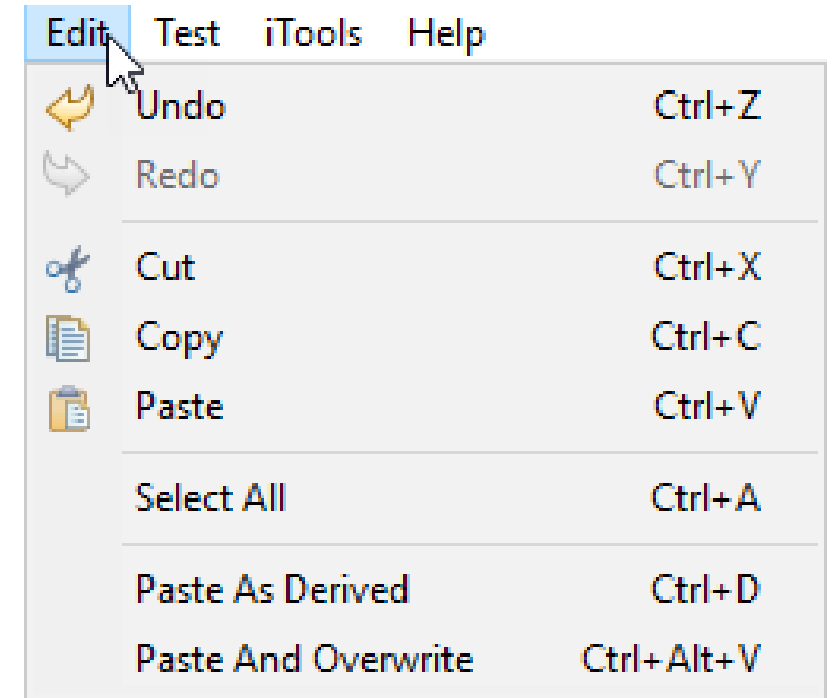
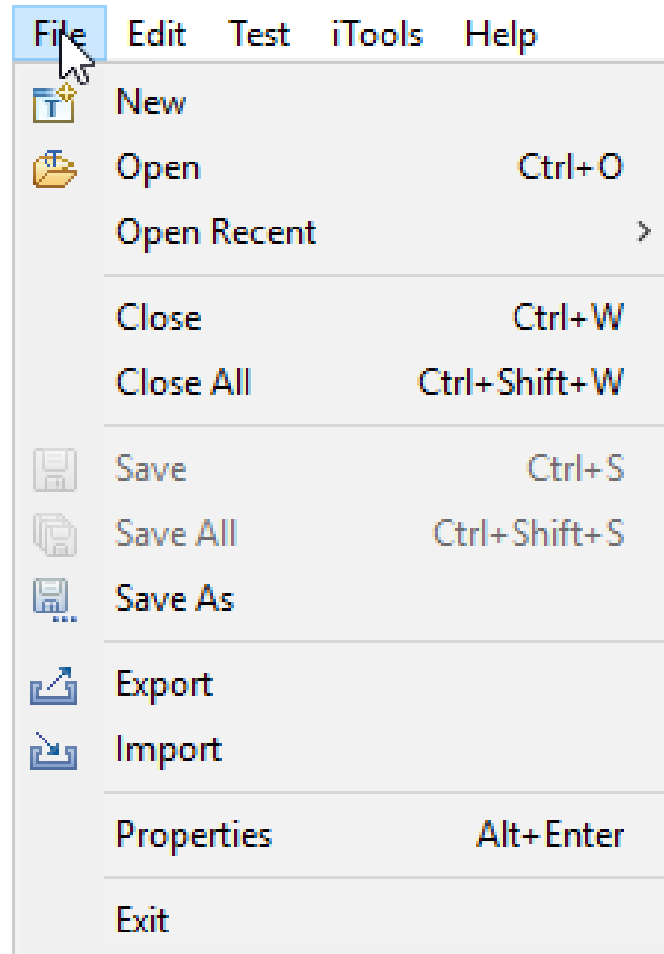
## Main menu bar

### File menu

The file menu contains the options to save \*.iYAML files, import and export options and the option to change testIDEA settings via the “Properties” dialogue.

### Edit menu

The edit menu offers the common edit options, such as copy and paste, as well as some testIDEA specific paste options.



# 4 FIRST STEPS WITH testIDEA



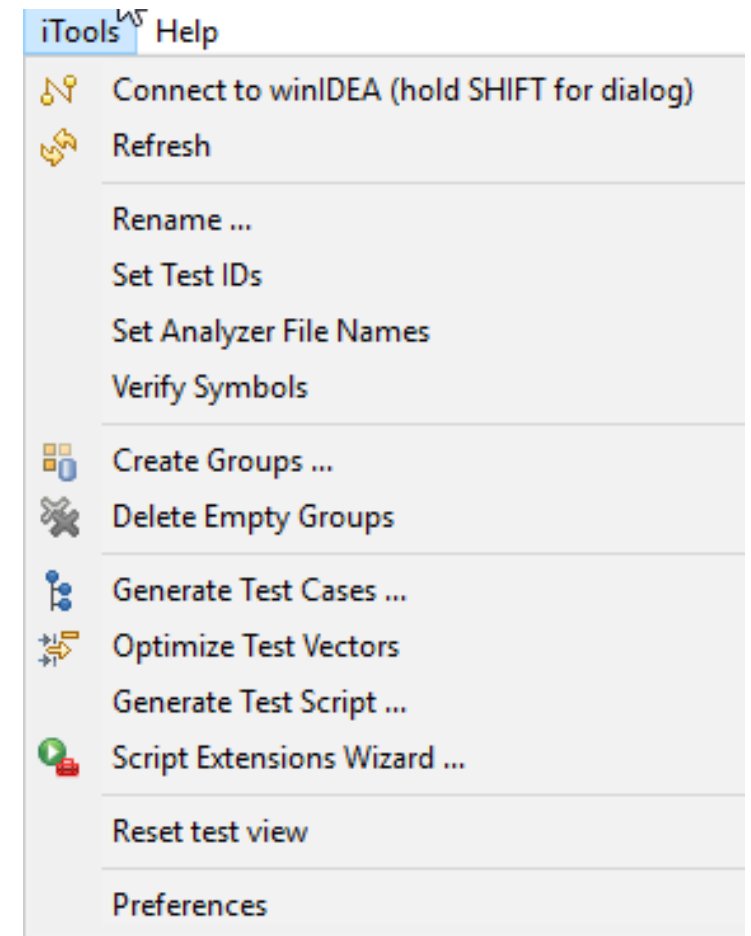
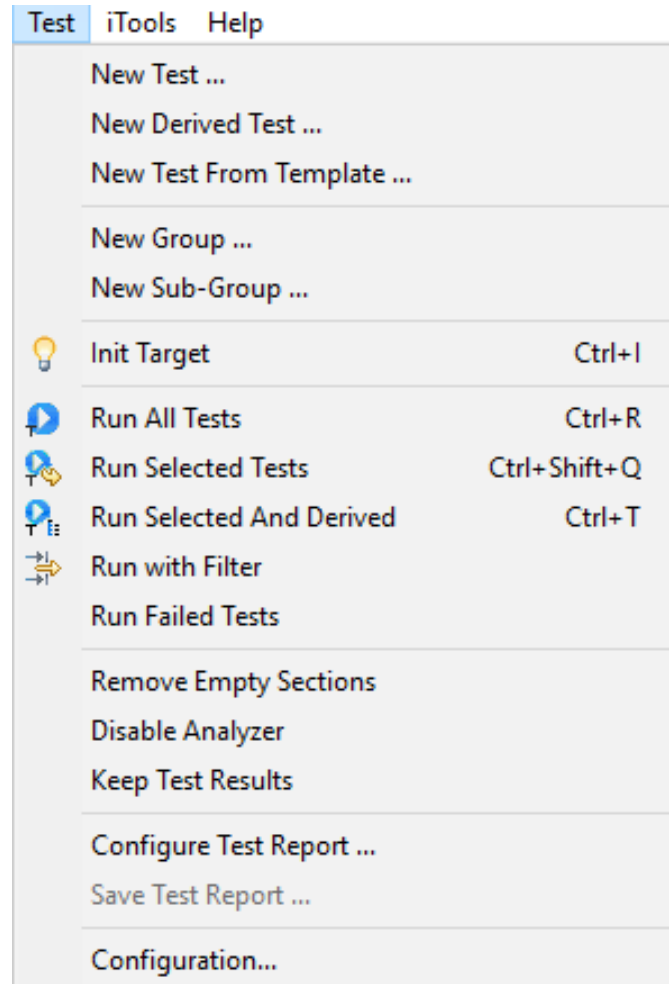
## Main menu bar

### Test menu

Via the test menu new tests can be created. Additionally, advanced grouping of tests can also be implemented. Further options allow all or selected tests to be executed, based upon selection or filter settings. Test reports can also be configured and created here.

### iTools menu

Via this menu, the connection to winIDEA and the project's symbols can be (re)established. Further advanced options are also offered here, along with access to testIDEA's preferences.



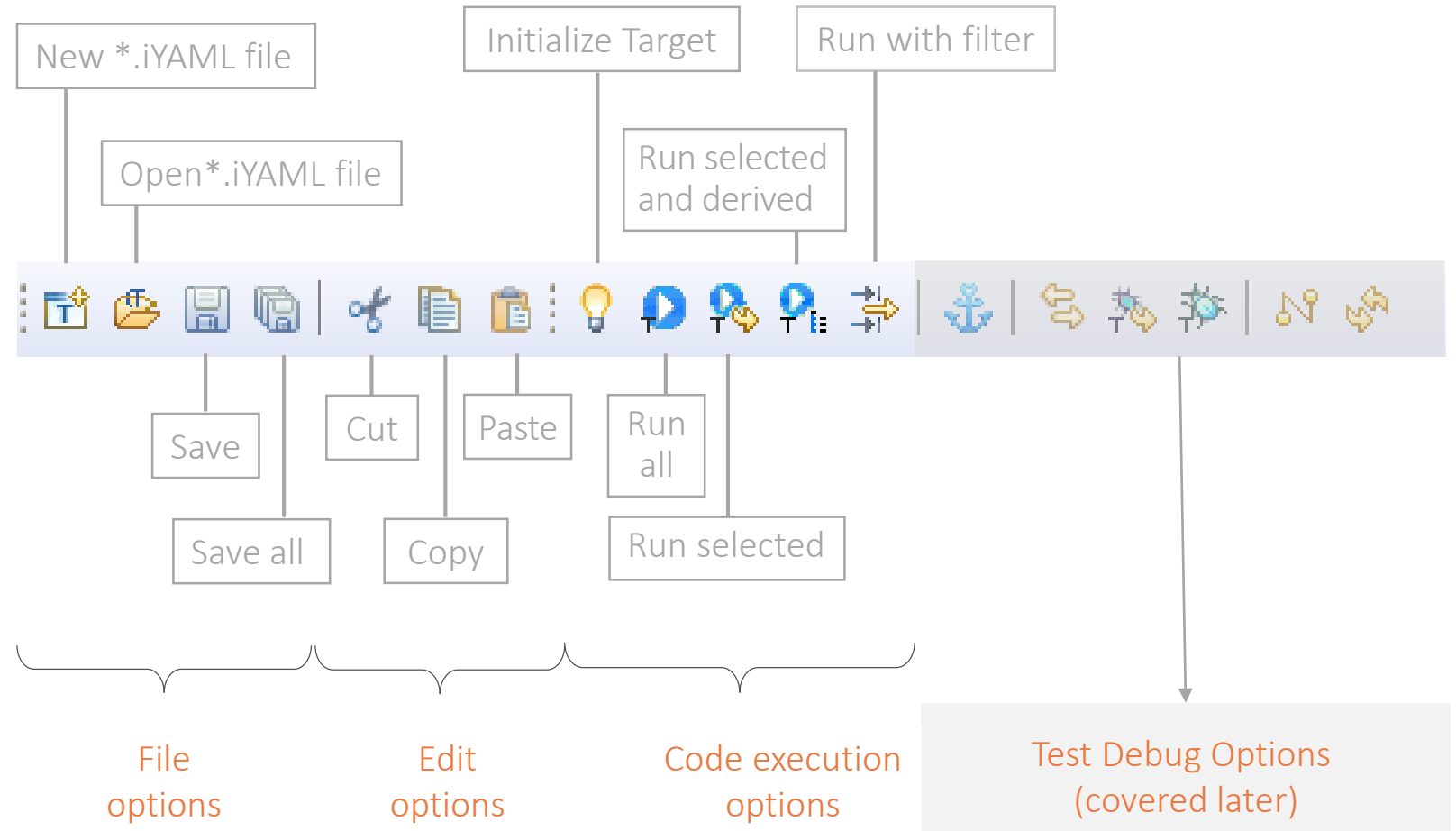
# 4 FIRST STEPS WITH testIDEA



## Main tool bar

The main tool bar offers some common file and edit options as well as various options for test execution, such as the option to run only selected test vectors. These options can be used to acquire quick test results or a merged coverage measurement for a select number of specific test vectors.

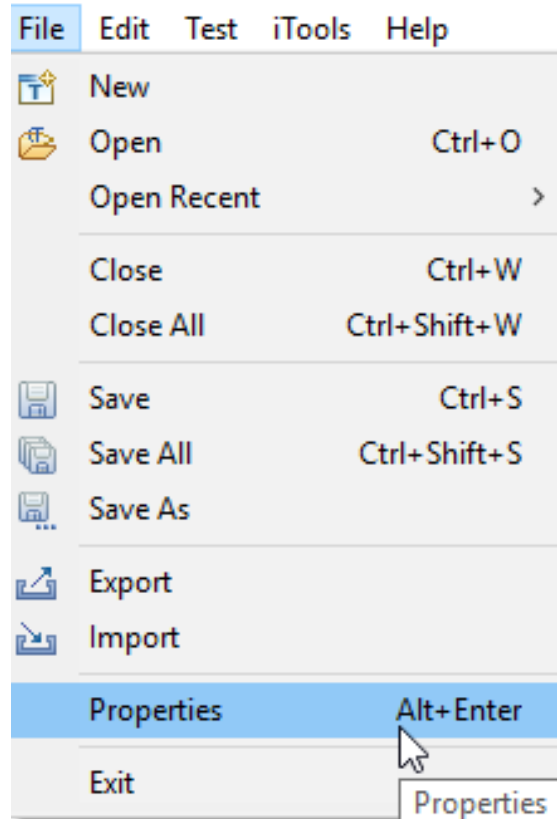
Furthermore, there are a few buttons for debug options that allow the code being tested to be debugged during the execution of the test. These options will be explained at the end of this unit.



# 5 SETUP THE TESTING ENVIRONMENT



Before execution of any tests, it is necessary to define some settings for the testing environment. This can be undertaken via the “*Properties*” option in the “*File*” menu.



[Skipping Setup Options](#)



*It is also possible to run test vectors without manually configuring the following setup options. In this case a window, suggesting use of the default settings, will appear when you start running your test vectors for the first time. It is recommended to accept the default settings if you did not make any changes in the “*Properties*” dialogue.*

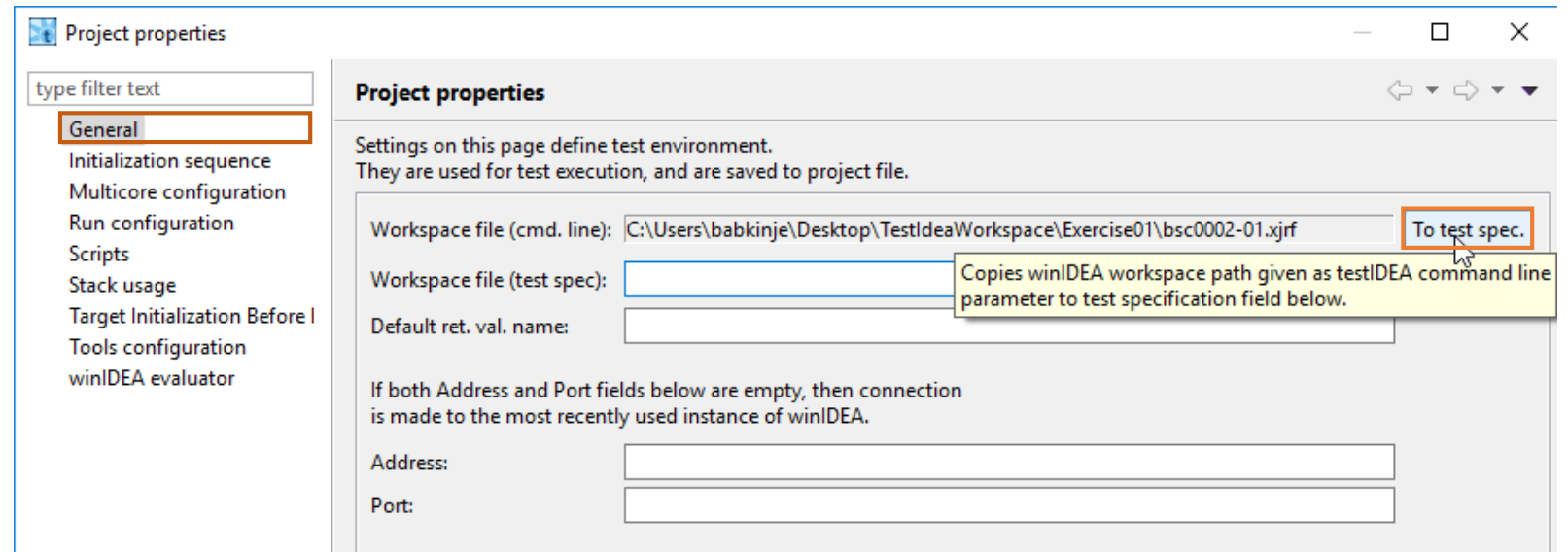


# 5 SETUP THE TESTING ENVIRONMENT



In the “*General*” settings, we can link the currently open winIDEA workspace to the testIDEA iYAML test specification file. This ensures that the test specification is linked to the winIDEA workspace to which it belongs whenever it is opened.

It is recommended to create the connection between winIDEA and testIDEA by using the “*To test spec.*” button. This ensures that the workspace with which testIDEA was started is associated with the tests.



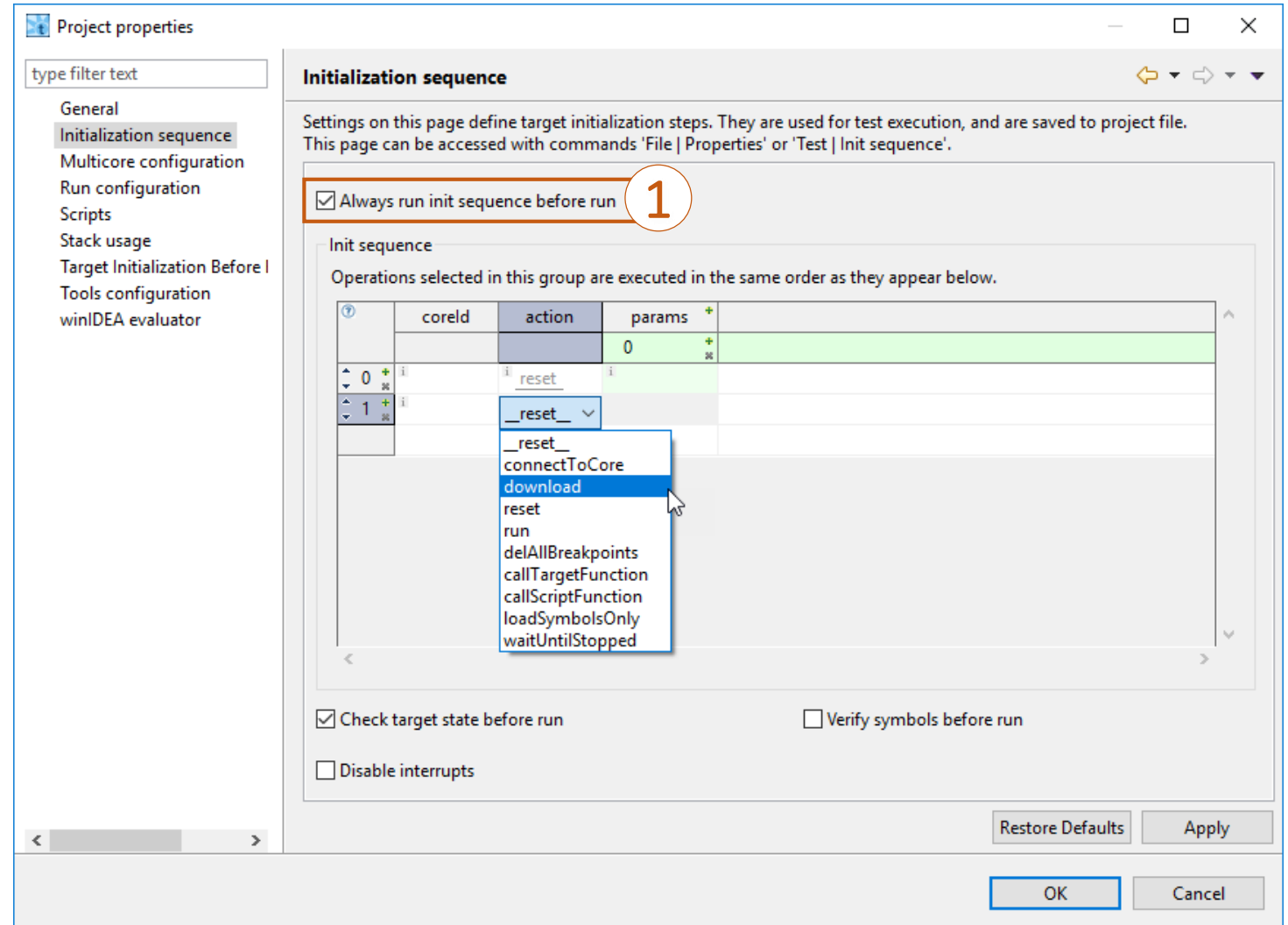
# 5 SETUP THE TESTING ENVIRONMENT



Set up the initialization sequence for the MCU. Typically the following process works in almost every case:

1

*“Always run init sequence before run” should be checked*

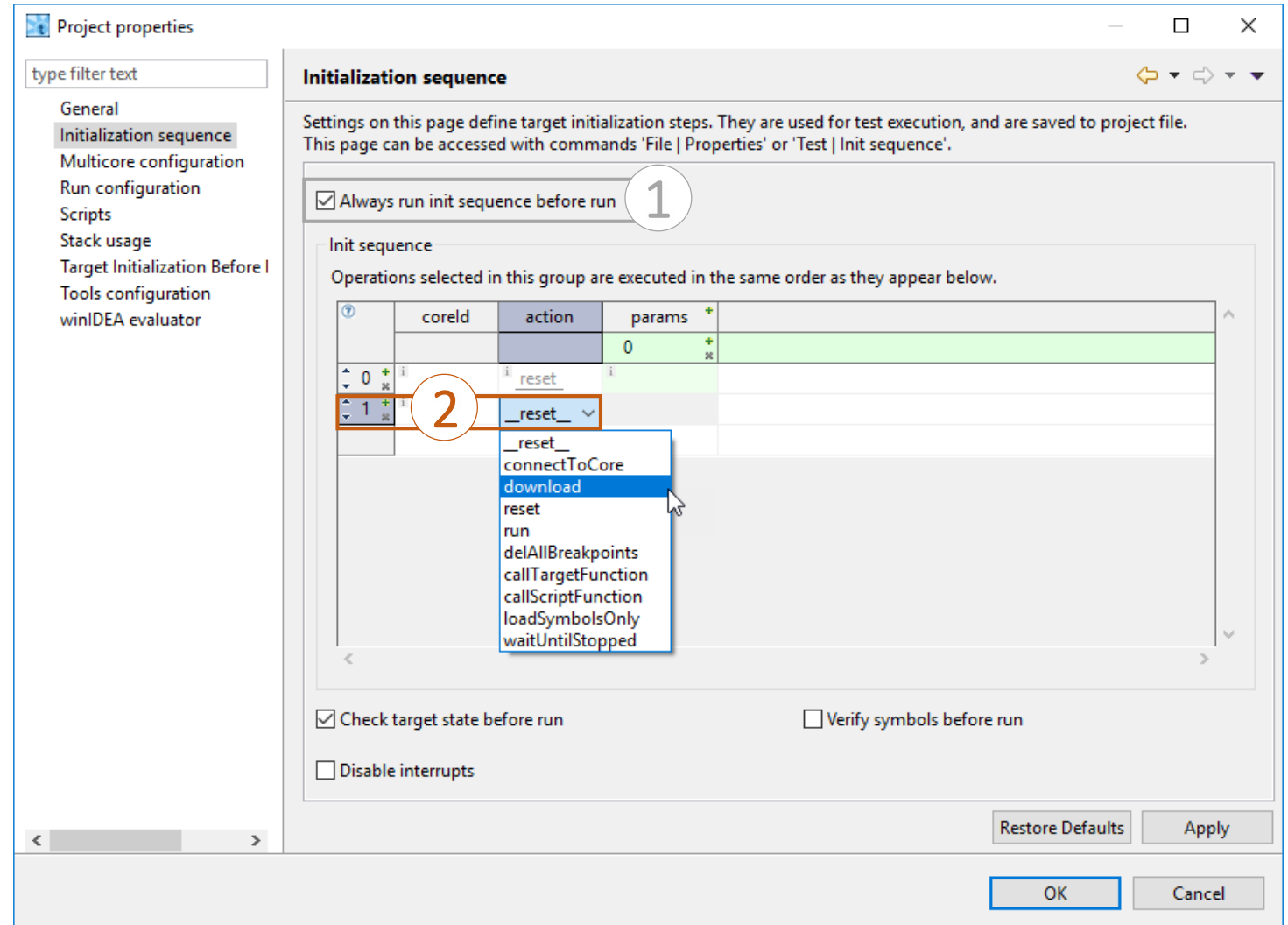


# 5 SETUP THE TESTING ENVIRONMENT



Set up the initialization sequence for the MCU. Typically the following process works in almost every case:

- 1 “Always run init sequence before run” should be checked
- 2 Click on the plus symbol and select a *reset* action

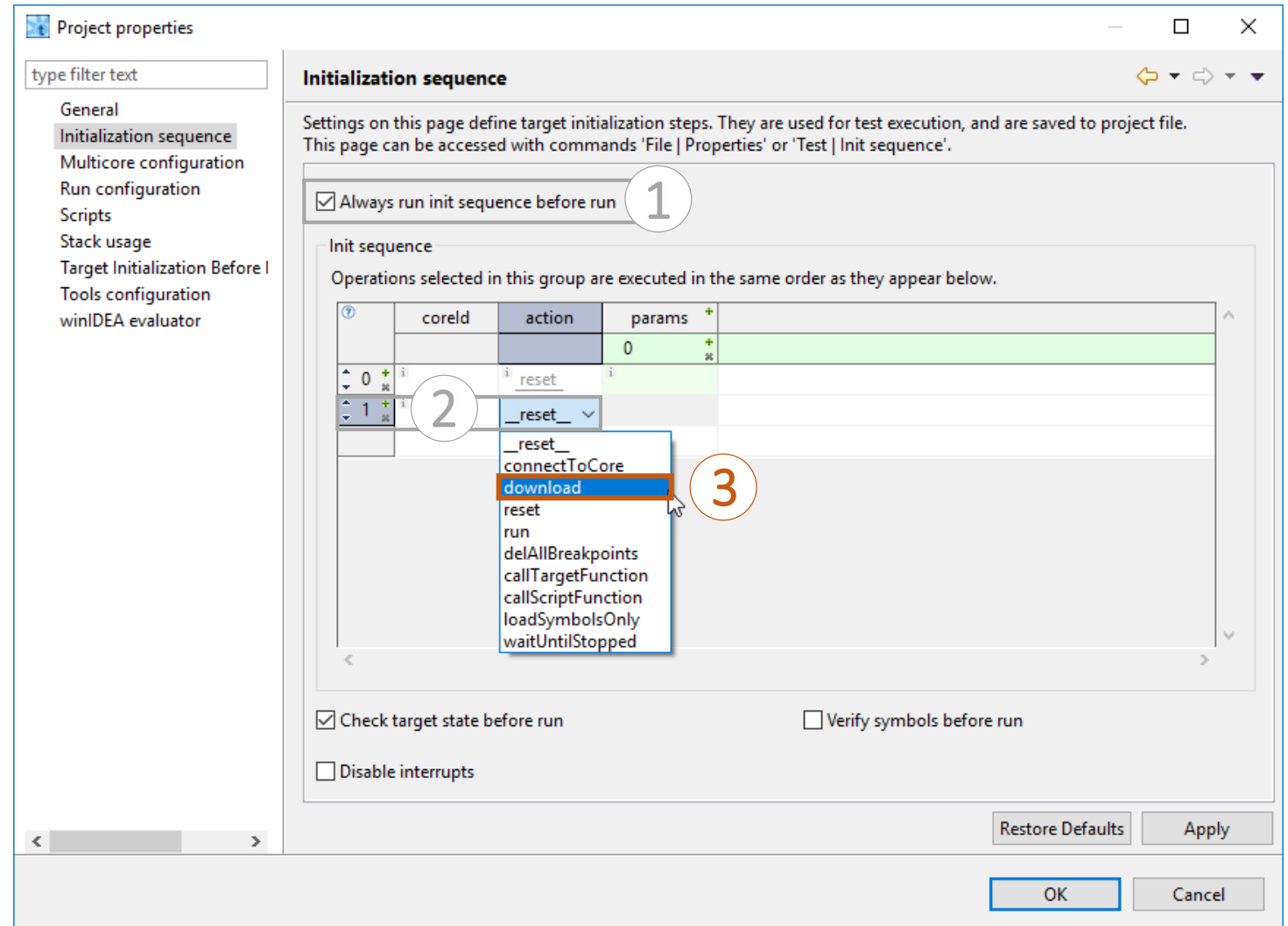


# 5 SETUP THE TESTING ENVIRONMENT



Set up the initialization sequence for the MCU. Typically the following process works in almost every case:

- 1 “Always run init sequence before run” should be checked
- 2 Click on the plus symbol and select a *reset* action
- 3 Modify the next action to be a *download* action

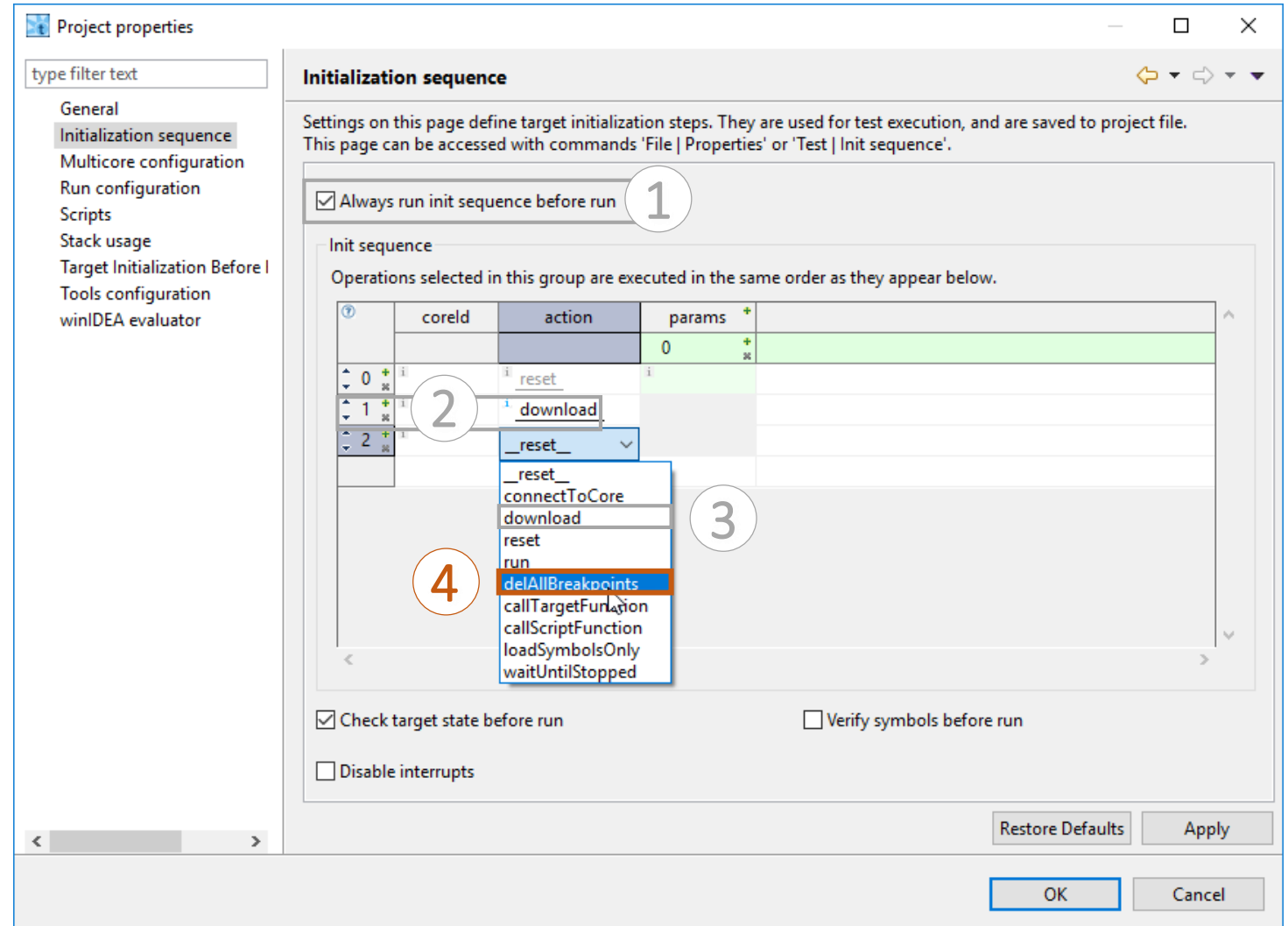


# 5 SETUP THE TESTING ENVIRONMENT



Set up the initialization sequence for the MCU. Typically the following process works in almost every case:

- 1 “Always run init sequence before run” should be checked
- 2 Click on the plus symbol and select a *reset* action
- 3 Modify the new action to a *download* action
- 4 Add another action to delete all breakpoints → *delAllBreakpoints*



# 5 SETUP THE TESTING ENVIRONMENT



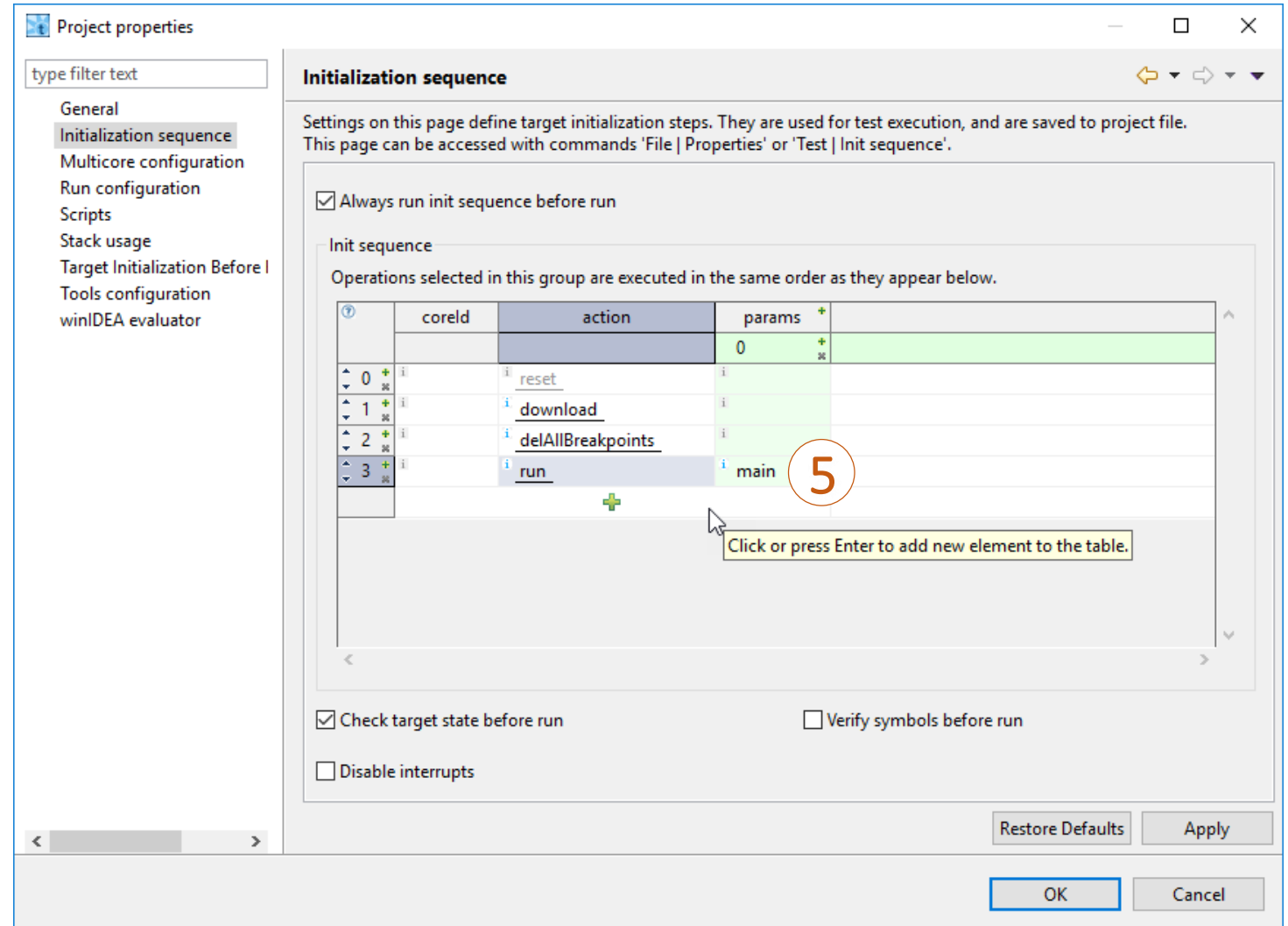
5

The last action is to execute the code on the target (→ *run* action) but only up to the entry of the **main()** function (*'main'* as *params.* means run until *main*).

Typically this is all that is required since, when the processor reaches the main function, the stack will have been initialized, a requirement for execution of original binary code tests on the target.



*If peripherals on the microcontroller also need to be instantiated prior to executing the tests, you may wish to name an alternate function to halt at here.*



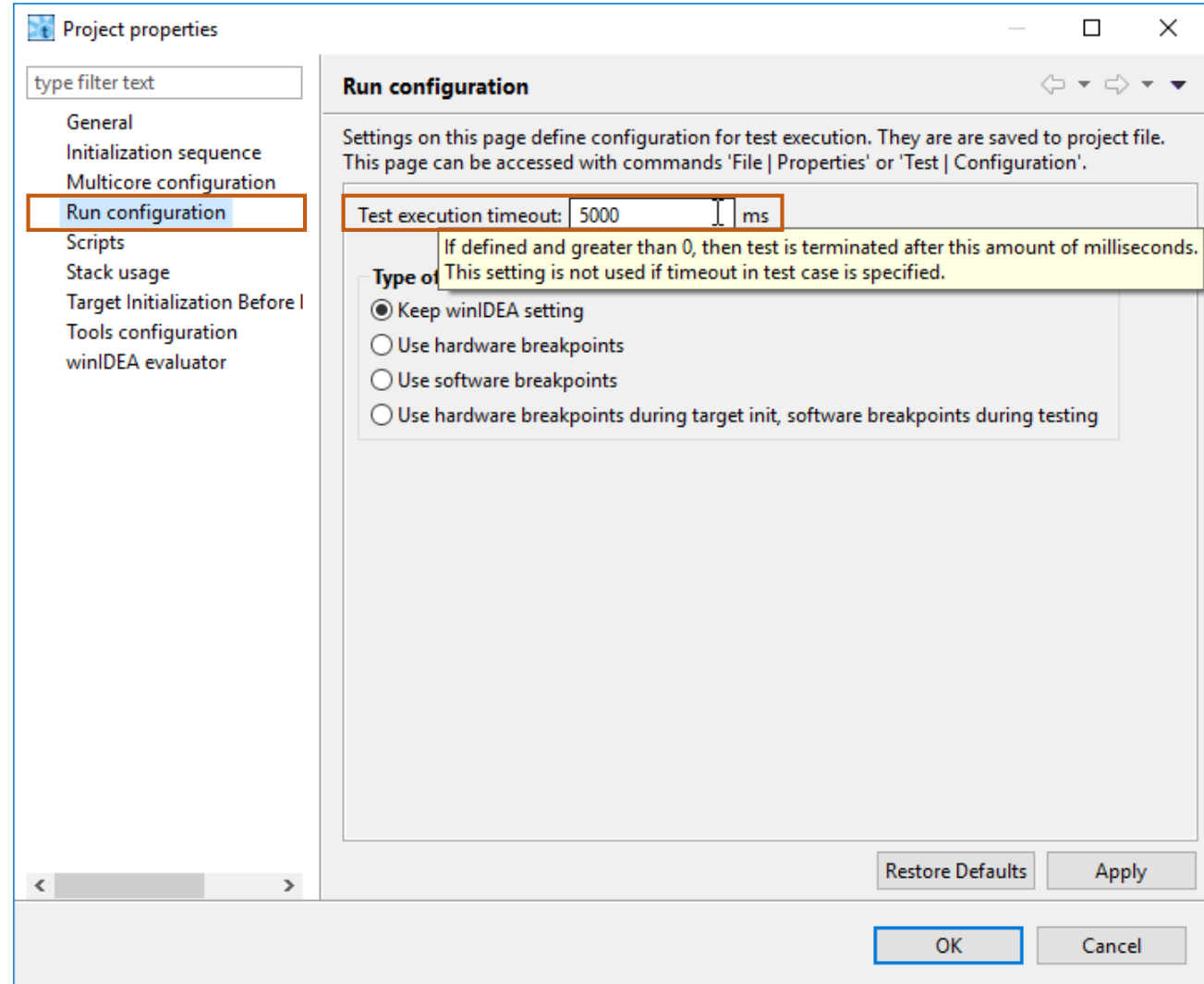
# 5 SETUP THE TESTING ENVIRONMENT



## Setup run configuration:

It is recommended to start by setting the test execution timeout to 5000ms.

A 5 second timeout means: in the event that the code on the target microcontroller hangs unexpectedly, perhaps it gets stuck in an infinite loop, after 5 seconds it will time out and provide an error, rather than hanging indefinitely.



# 6 CREATE A NEW BASE TEST - THE APPLICATION

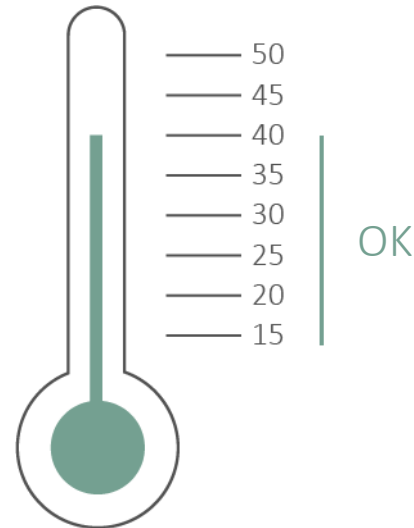


Take a look at the following source code:

The function that we want to create unit tests for is named `evaluateTemperature()`, a simple function that accepts a single parameter by value and delivers a single return parameter by value. The return value is of type `Thermostat`, and is limited to four possible values.

During code development it was decided to create a new data type using `enum` rather than define the values using `#define`. This conscious decision simplifies test creation since testIDEA will find the `Thermostat` type automatically in the binary file's symbols.

```
enum Thermostat {  
    TEMP_ERROR,  
    TEMP_UNDER_15,  
    TEMP_OK,  
    TEMP_OVER_40  
};
```



*By sticking to an agreed coding standard (such as using an enum rather than #define as seen here), the resulting code becomes easier to debug and maintain.*

```
Thermostat thermostateControl =  
    TEMP_ERROR;  
Thermostat evaluateTemperature(  
    signed int temperature) {  
  
    if (temperature < 15) {  
        returnValue = TEMP_UNDER_15;  
    } else if (temperature <= 40) {  
        returnValue = TEMP_OK;  
    } else {  
        returnValue = TEMP_OVER_40;  
    }  
  
    return returnValue;  
}
```



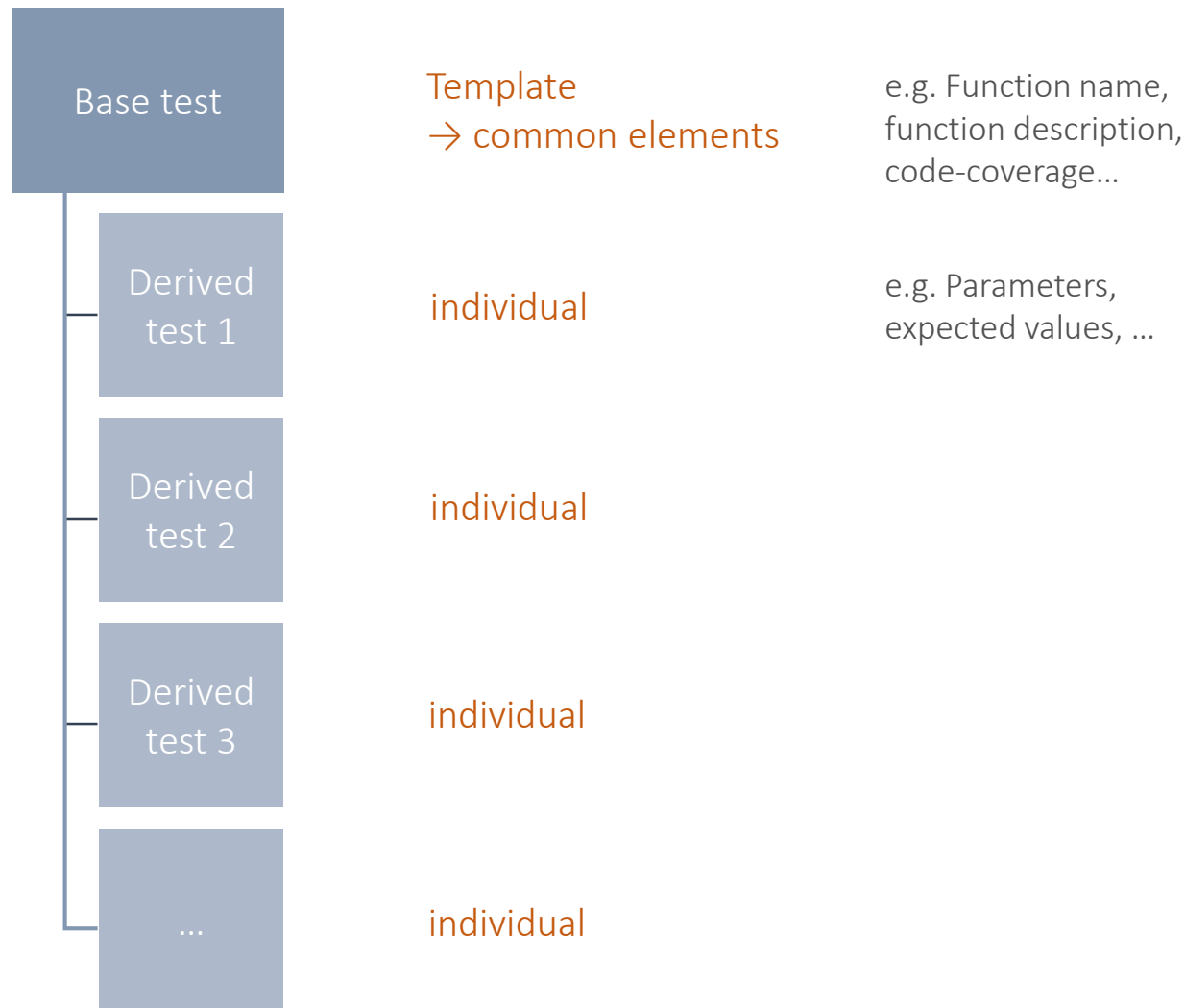
# 6 CREATE A NEW BASE TEST - CONCEPT

## Set up of the testing environment

We will start by creating a base test which we do not want to actually execute. This is a template for the tests we wish to create.

We will then derive further tests from the base test.

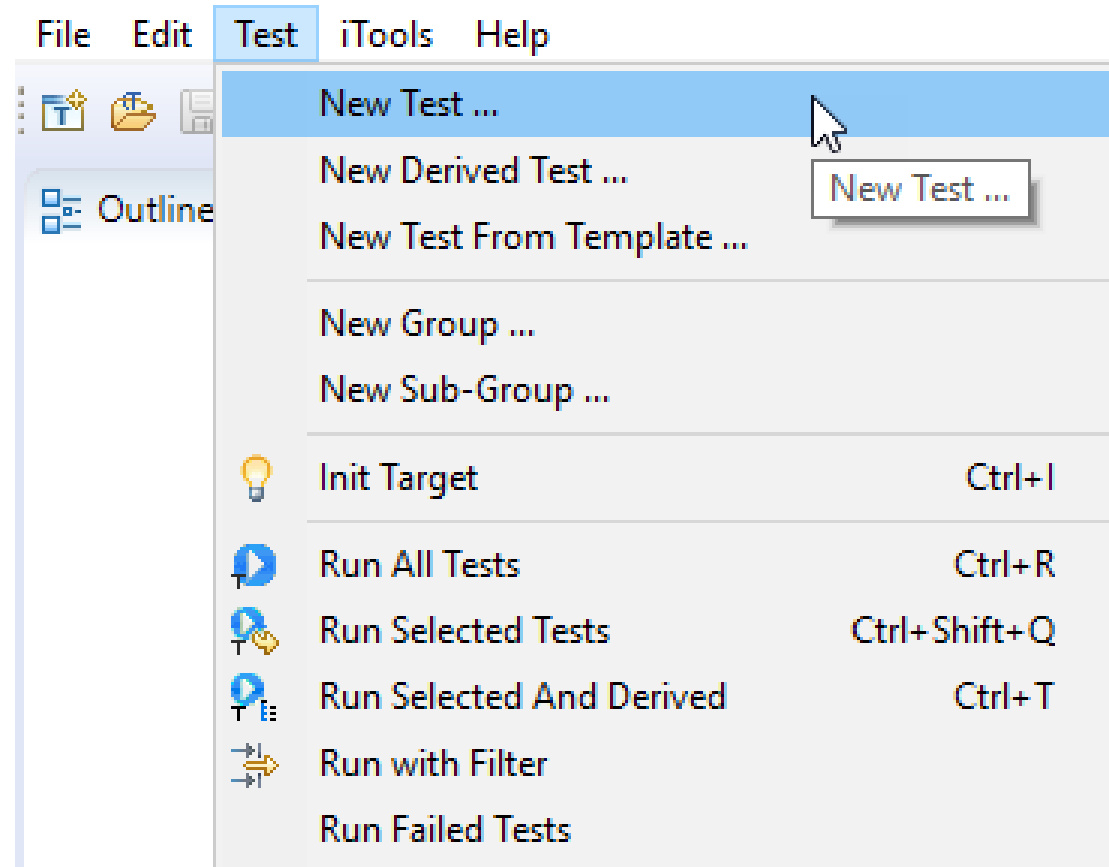
The base test can be imagined as a template containing all of the key information for testing a C function or a C++ method that is common to all of the tests. Anything that is specific to each individual test (such as input parameters or return values) is left out.



# 6 CREATE A NEW BASE TEST



A new base test is created by using the “Test” menu in the main menu bar and selecting “New Test...”



# 6 CREATE A NEW BASE TEST



When we first click on the function drop down menu there is nothing available in terms of function names for testing. This is because this new instantiation of testing isn't currently connected to a winIDEA project. Clicking on the Refresh button will establish this connection and all the symbol information from the binary file will be transferred to testIDEA.

Using the Refresh Button will also change the state of testIDEA

EVALUATION

# 6 CREATE A NEW BASE TEST



1

When we now open the drop-down menu, all of the functions that are included in the symbols of the associated binary file are listed. It is now possible to select the function called *evaluateTemperature()* from this list.

**New test case wizard**

Enter basic test case information. Button 'Next' is enabled only for unit tests if function name is defined and symbols are loaded.

Scope: ☐ Unit ☐ System ☒ Default (Unit) ☒ Auto generate test ID

Core ID:

Function:  1

Parameters:

**Expected result**

☒ Default expression for function return value test  
\_isys\_rv ==

☐ Custom expression and function return value name  
Expression:   
Ret. val. name:

< Back Next > **Finish** Cancel

# 6 CREATE A NEW BASE TEST



1 When we now open the drop-down menu, all of the functions that are included in the symbols of the associated binary file are listed. It is now possible to select the function called *evaluateTemperature()* from this list.

2 The return type and the type and variables associated with the parameter list are then automatically filled in the field below as this information has been extracted from the binary file.

New test case wizard

New test case wizard

Enter basic test case information. Button 'Next' is enabled only for unit tests if function name is defined and symbols are loaded.

Scope: ☐ Unit ☐ System ☒ Default (Unit) ☒ Auto generate test ID

Core ID:

Function:  1

2  Name of a C function, which we want to test.

Parameters:

**Expected result**

☒ Default expression for function return value test

☐ Custom expression and function return value name

Expression:

Ret. val. name:

< Back Next > Finish Cancel

# 6 CREATE A NEW BASE TEST



- 3 This is all we have to do for this particular test because the only common element for further derived unit tests will be the name of the function. The parameter and the expected return value will be test dependent, so we leave these fields empty for now.

New test case wizard

**New test case wizard**

Enter basic test case information. Button 'Next' is enabled only for unit tests if function name is defined and symbols are loaded.

Scope: ☐ Unit ☐ System ☒ Default (Unit) ☒ Auto generate test ID

Core ID:

Function:  1

2

Parameters:  3

**Expected result**

☒ Default expression for function return value test

☐ Custom expression and function return value name

Expression:

Ret. val. name:

# 6 CREATE A NEW BASE TEST



3 This is all we have to do for this particular test because the only common element for further derived unit tests will be the name of the function. The parameter and the expected return value will be test dependent, so we leave these fields empty for now.

4 Click “Finish” and you will have created your first base test.

New test case wizard

New test case wizard

Enter basic test case information. Button 'Next' is enabled only for unit tests if function name is defined and symbols are loaded.

Scope: ☐ Unit ☐ System ☒ Default (Unit) ☒ Auto generate test ID

Core ID:

Function:  1

2  Name of a C function, which we want to test.

Parameters:  3

Expected result

☒ Default expression for function return value test

☐ Custom expression and function return value name

Expression:

Ret. val. name:

3

4

< Back Next > Finish Cancel

# 6 CREATE A NEW BASE TEST



## The *Meta* form:

We will now clear the “*Execute*” box in the “*Meta*” section as we do not want to execute this base test, as there are no parameters and no expected return values in this test template.

The screenshot shows the 'Meta' form in the testIDEA application. The left sidebar contains a tree view with the following items: Meta (selected), Function, Persistent variables, Variables, Pre-conditions, Expected, Stubs, User Stubs, Test Points, Analyzer, Coverage, Statistics, Profiler, Code areas, Data areas, Trace, HIL, Scripts, and Options. The main area of the form has the following fields: 'Execute' (checked), 'Scope' (empty), 'ID' (empty), 'Description' (empty), and 'Result comment' (empty). A tooltip is visible over the 'Execute' checkbox, stating: 'Check this box to enable the test. If unchecked, test will not be executed regardless of filters. It should be unchecked for test specifications, which are used as base for derived exceptions only, and are not intended for execution.' There are also 'Inherit' checkboxes for 'Scope' and 'ID'.

This is a close-up of the 'Execute' checkbox and its tooltip. The checkbox is checked. The tooltip text reads: 'Check this box to enable the test. If unchecked, test will not be executed regardless of filters. It should be unchecked for test specifications, which are used as base for derived exceptions only, and are not intended for execution.'



# 6 CREATE A NEW BASE TEST



## The *Function* form:

Looking at the function form, we can check that the inputs from the prior steps are correctly displayed.

The screenshot shows the 'Function' form in the testIDEA application. The left sidebar contains a tree view with the following items: Meta, Function (highlighted with a red box), Persistent variables, Variables, Pre-conditions, Expected, Stubs, User Stubs, Test Points, Analyzer (expanded), Coverage (expanded), Statistics, Profiler (expanded), Code areas, Data areas, Trace, HIL, Scripts, and Options. The main area is titled 'Function' and contains the following fields:

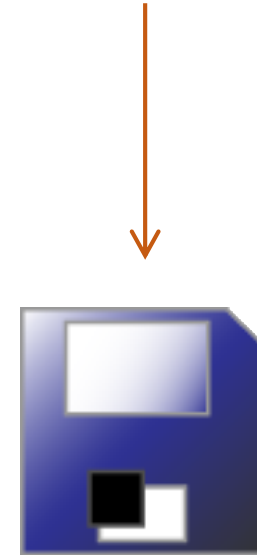
- Function:** A text field containing 'evaluateTemperature'.
- Params:** A text field containing 'Thermostat (long temperature)'.
- Ret. val. name:** A text field.
- Test exec. timeout:** A text field with a unit 'ms'.

Each of these fields has an 'Inherit' checkbox to its left. The 'Form' tab is selected at the bottom of the window.

## 6 CREATE A NEW BASE TEST



After completing the first base test it is recommended to save the file.



# 7 CREATE A DERIVED TEST - CONCEPT

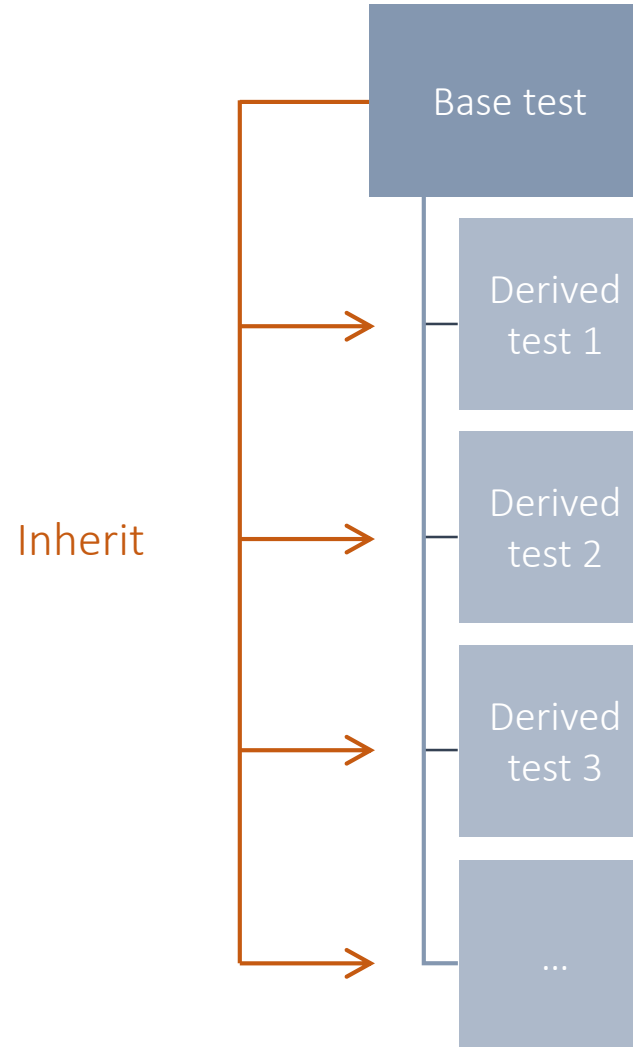


## Derived tests:

- Tests are organized hierarchically in a tree structure
- Tests at the top level are called **base tests** (parent) while their children are called **derived tests**
- Derived tests can **inherit** traits from lower level tests



*Derived tests may override settings of the base test if required.*



# 7 CREATE A DERIVED TEST - HEREDITY



Basically, anything that exists in a base test will also be copied to the derived test.

If a field was filled with information, it appears filled in the derived test.

If a field was unfilled, it remains unfilled in the derived test.

On the right we have an example of a base test (top) and the derived test (bottom). The derived test has inherited the function *ECLIB\_Sqr\_16* and, as a result, it is grayed out (in blue) to protect it from change.

The screenshot displays two windows from the testIDEA application, illustrating the inheritance of test configuration from a base test to a derived test.

**Base Test Window:** The title bar reads "bsc0002-03-test-vectors-imported-and-persistent.iyam". The left sidebar shows a tree view with categories like Meta, Function, Persistent variables, Variables, Pre-conditions, Expected, Stubs, User Stubs, Test Points, Analyzer, Coverage, Profiler, and Trace. The main area shows the "Function:" field with "ECLIB\_Sqr\_16" selected. Below it, the "Params:" field is empty. The "Inherit" checkbox is checked.

**Derived Test Window:** The title bar reads "\*bsc0002-03-test-vectors-imported-and-persistent.iyam". The left sidebar is identical to the Base Test window. The main area shows the "Function:" field with "ECLIB\_Sqr\_16" selected. Below it, the "Params:" field is empty. The "Ret. val. name:" field is empty. The "Test exec. timeout:" field is empty. The "Core ID:" field is empty. The "Inherit" checkbox is checked. The entire content area of the Derived Test window is grayed out (in blue), indicating that the configuration is inherited from the Base Test and cannot be modified.

# 7 CREATE A DERIVED TEST - HEREDITY



As there were no parameters entered into the base test, there are no parameters in the derived test. The related field remains white and it is possible to enter values.

The screenshot displays two overlapping windows from the testIDEA software. The top window, titled 'bsc0002-03-test-vectors-imported-and-persistent.iyam', is labeled 'Base Test'. It shows a tree view on the left with 'Function' selected. The main area has 'Function: ECLIB\_Sqr\_16' and an unchecked 'Inherit' checkbox. The bottom window, titled '\*bsc0002-03-test-vectors-imported-and-persistent.iyam', is labeled 'Derived Test'. It also shows the 'Function' selected in the tree view. The main area has 'Function: ECLIB\_Sqr\_16' and a checked 'Inherit' checkbox. An orange arrow points from the 'Inherit' checkbox in the 'Derived Test' window to the 'Inherit' checkbox in the 'Base Test' window, illustrating the inheritance of settings.

# 7 CREATE A DERIVED TEST - HEREDITY



As there were no parameters entered into the base test, there are no parameters in the derived test. The related field remains white and it is possible to enter values.

The *Inherit* setting refers to the elements the **derived test** inherits from the **test from which it is derived** (in this case, our base test).

In both cases, the *Inherit* setting is set to *intermediate* in the derived test, the default setting. This is indicated by the black check box. The intermediate setting indicates that the field will inherit entries from the test from which it was derived, protecting them from change, unless the field was empty, leaving the field open for editing.

The screenshot displays two windows from the testIDEA application, illustrating the configuration of a derived test based on a base test.

**Base Test Window:**

- Function:** ECLIB\_Sqr\_16
- Params:** (Empty)
- Inherit:** Checked (indicated by a black check box)

**Derived Test Window:**

- Function:** ECLIB\_Sqr\_16
- Params:** (Empty)
- Ret. val. name:** (Empty)
- Test exec. timeout:** (Empty)
- Core ID:** (Empty)
- Inherit:** Checked (indicated by a black check box) for the Function, Params, Ret. val. name, Test exec. timeout, and Core ID fields.

# 7 CREATE A DERIVED TEST - HEREDITY



The Inherit setting can be set to:

- Unchecked: to explicitly not inherit entries from the base test.
- Intermediate: inherit and protect entries derived from the base test, if there; otherwise leave empty.
- Checked: explicitly inherit the setting from the base test.



*Always develop the base test using the lowest common denominator of settings. Exceptions to the common denominator can then be edited by hand for the few outliers in your derived tests.*

The screenshot displays two overlapping windows from the testIDEA software, both titled 'bsc0002-03-test-vectors-imported-and-persistent.iyam'. The top window, labeled 'Base Test', shows a configuration panel with a tree view on the left containing categories like Meta, Function, Persistent variables, Variables, Pre-conditions, Expected, Stubs, User Stubs, Test Points, Analyzer, Coverage, Statistics, Profiler, Code areas, Data areas, Trace, HIL, Scripts, Options, Dry run, and Diagrams. The main panel has an 'Inherit' checkbox checked, a 'Function' dropdown set to 'ECLIB\_Sqr\_16', and an empty 'Params' field. The bottom window, labeled 'Derived Test', shows the same configuration panel. In this window, the 'Inherit' checkbox is checked, the 'Function' dropdown is set to 'ECLIB\_Sqr\_16', the 'Params' field is empty, the 'Ret. val. name' field is empty, and the 'Test exec. timeout' field is set to 'ms'. The 'Inherit' checkbox for the 'Test exec. timeout' field is also checked. The 'Core ID' field is empty.

# 7 CREATE A DERIVED TEST – FIND PARAMETERS AND EXPECTED VALUES



We want to test `evaluateTemperature()` using a boundary testing strategy (boundaries of the data types are not considered in this example):

*signed int temperature = 14*

*signed int temperature = 15*

*signed int temperature = 16*

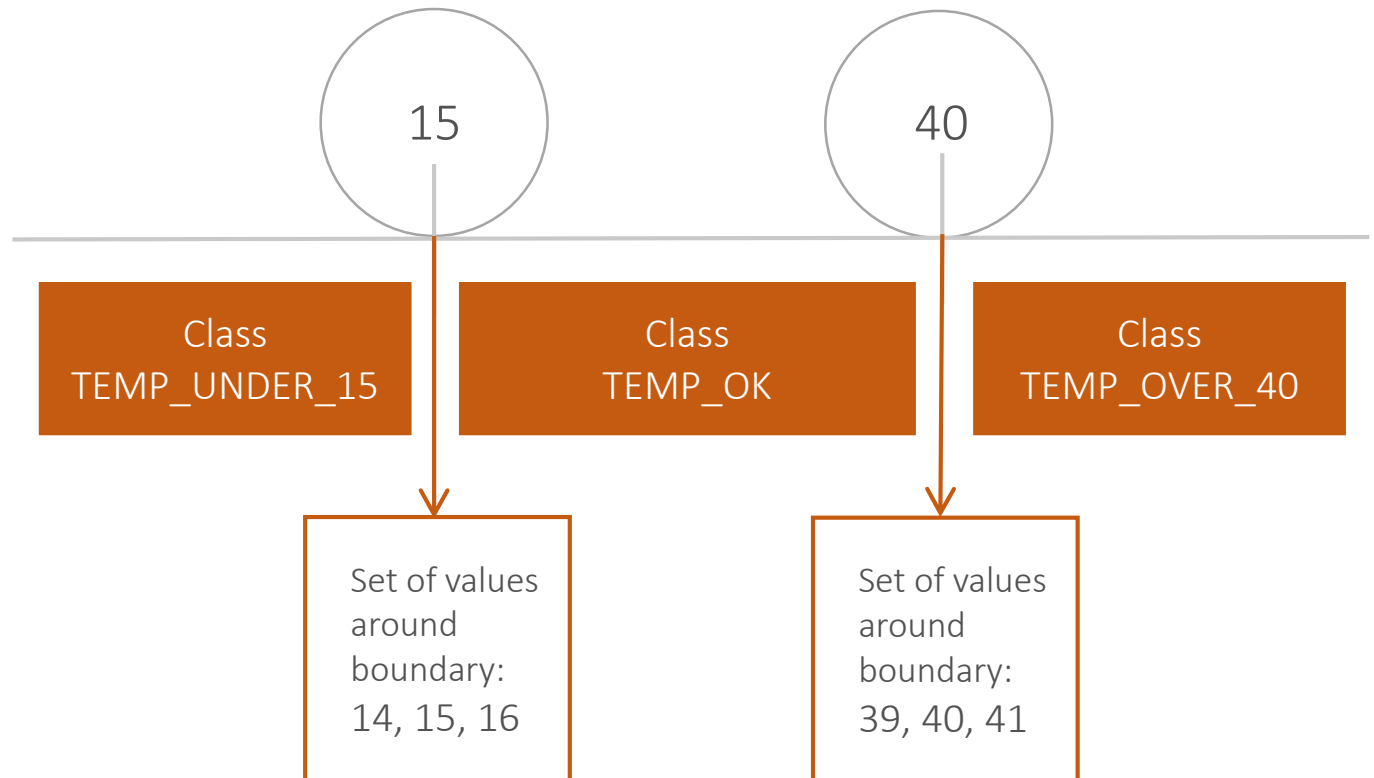
*signed int temperature = 39*

*signed int temperature = 40*

*signed int temperature = 41*

The created base test will be used to create **derived tests**.

```
if (temperature < 15) {  
    returnValue = TEMP_UNDER_15;  
} else if (temperature <= 40) {  
    returnValue = TEMP_OK;  
} ...  
...  
} else {  
    returnValue = TEMP_OVER_40;  
}  
return returnValue;
```

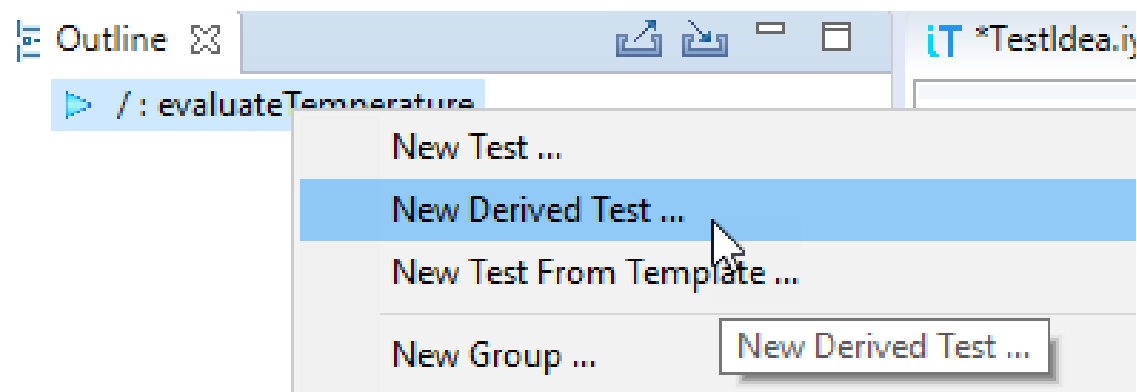




# 7 CREATE A DERIVED TEST

To create a new derived test start by selecting the base test we created with the mouse. Then open the context menu (right mouse click) in the *Outline* from the base test and select “*New Derived Test...*”.

The *New derived test case wizard* will open.



# 7 CREATE A DERIVED TEST



1

We do not need to enter anything into the function field, because this test inherits the information from the base test.

New derived test case wizard

**New test case wizard**

Enter basic test case information. Button 'Next' is enabled only for unit tests if function name is defined and symbols are loaded.

Scope: ☐ Unit ☐ System ☒ Default (Unit) ☒ Auto generate test ID

Core ID:

Function:  1

Parameters:  15

Function parameters, for example: 10, 30, 'c'

**Expected result**

☐ Default expression for function return value test

☒ Custom expression and function return value name

Expression:

Ret. val. name:

< Back Next > Finish Cancel

# 7 CREATE A DERIVED TEST



- 1 We do not need to enter anything into the function field, because this test inherits the information from the base test.
- 2 We can now add a test input parameter; in this example we choose 15°C as a parameter value.

New derived test case wizard

**New test case wizard**

Enter basic test case information. Button 'Next' is enabled only for unit tests if function name is defined and symbols are loaded.

Scope: ☐ Unit ☐ System ☒ Default (Unit) ☒ Auto generate test ID

Core ID:

Function:

Parameters:   Function parameters, for example: 10, 30, 'c'

**Expected result**

☐ Default expression for function return value test

☒ Custom expression and function return value name

Expression:

Ret. val. name:

< Back Next > Finish Cancel

# 7 CREATE A DERIVED TEST



3

Next we have to define the expected return value for this test parameter. In the case of a 15°C input value we expect TEMP\_OK as the return value as it lies in the 15° to 40° range programmed in the code. `_isys_rv` is a default return value variable that testIDEA creates for us in order to capture any return value that results from tested functions.

In the *Expected result* field we can enter any valid C/C++ evaluation expression using test or target variables, registers, or I/O module input ports.

The returned value is then verified against this expected value or expression.

# 7 CREATE A DERIVED TEST



4

Clicking “Finish” will create your first derived test



Save some time typing by simply copying the return value text from the source code in the winIDEA editor window and pasting it into the “Expected result” field.

New derived test case wizard

**New test case wizard**

Enter basic test case information. Button 'Next' is enabled only for unit tests if function name is defined and symbols are loaded.

Scope: ☐ Unit ☐ System ☒ Default (Unit) ☒ Auto generate test ID

Core ID:

Function:  1

Parameters: 15 2

**Expected result**

☒ Default expression for function return value test

\_isys\_rv == TEMP\_OK 3

☐ Custom expression and function return value

Expression:

Ret. val. name:

Enter expected function return value. This value will be used to automatically generate expression '\_isys\_rv == <value>' in section 'Expected'. For example, if you enter: 10 expression '\_isys\_rv == 10' will be automatically generated. This feature can only be used for scalar types (char, int, ...). For complex types specify Ret. val. name and expression below. Additional expressions can later be entered in section 'Variables'.

< Back Next > Finish 4 Cancel

# 7 CREATE A DERIVED TEST - FUNCTION DATA



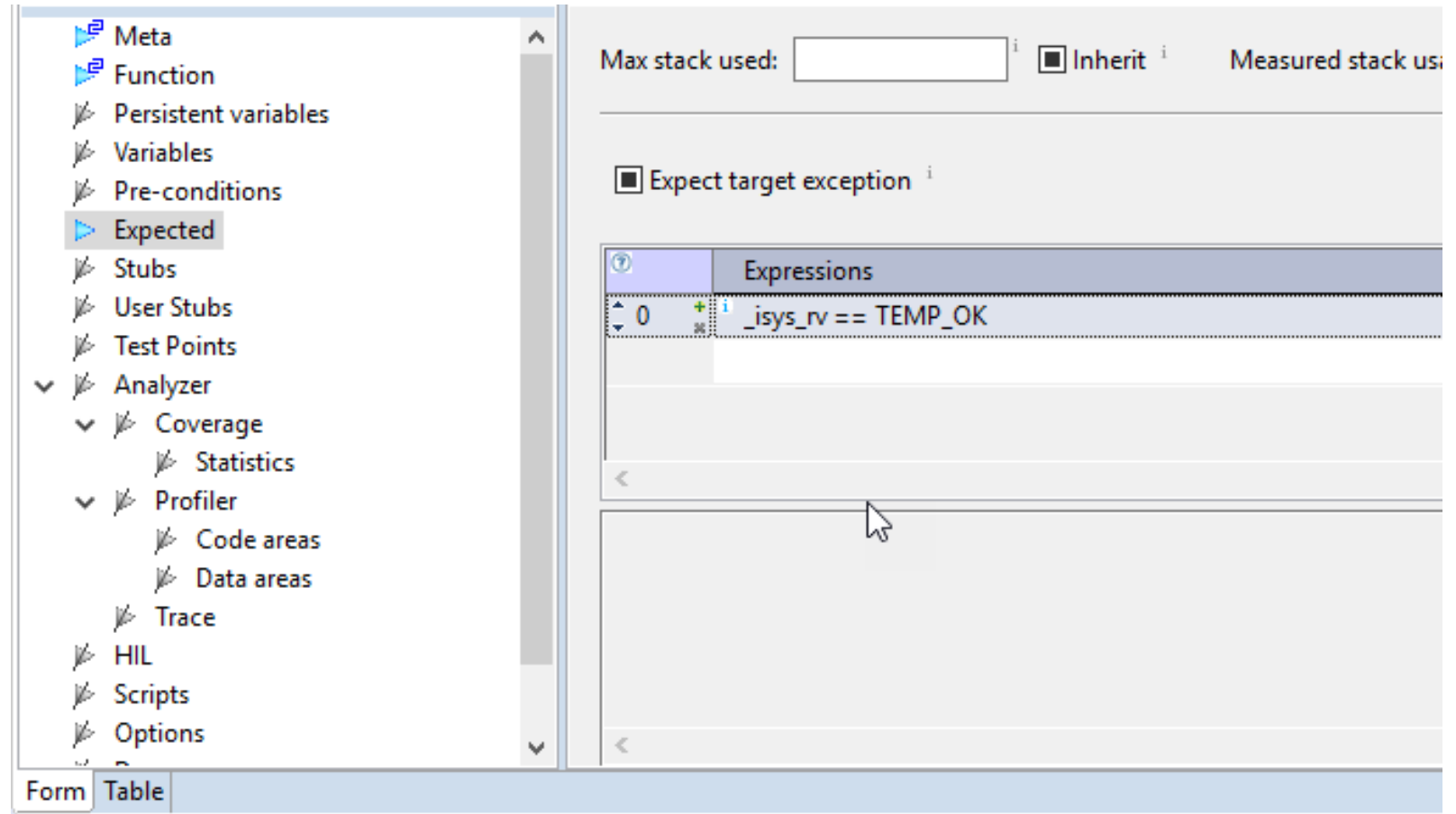
In the test case's options list you can now see that some of the arrows turned blue. These blue arrows indicate options with values entered into them. The blue arrow with the small backward symbol indicates some of the fields contain inherited elements.

Selecting the *Function* form, we see that the *Function* field and other blue colored elements are inherited entries, whilst the *Parameter* field is unique to this test case.

# 7 CREATE A DERIVED TEST - EXPECTED VALUE DATA



Selecting the *Expected* values view we see that our expected value “TEMP\_OK” is listed here as the comparison for the returned result.

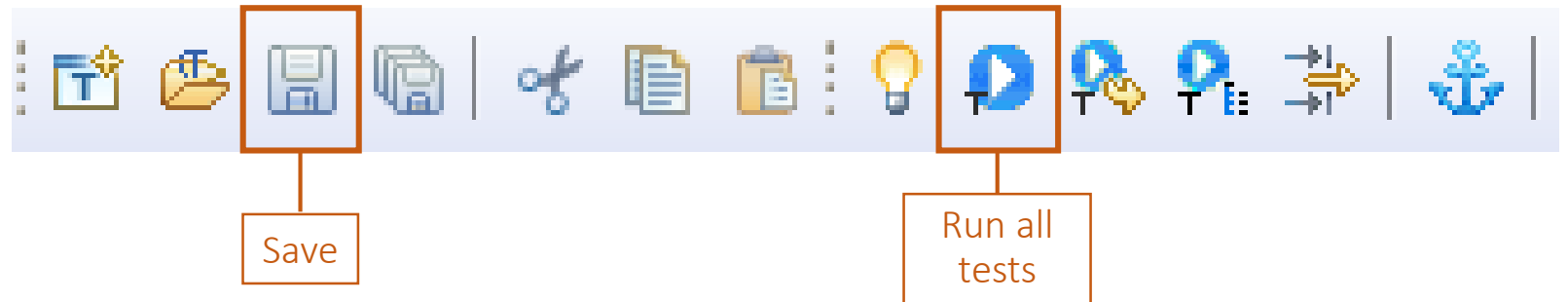


# 7 CREATE A DERIVED TEST - RUN



The *Save* icon can be used to save the updated test specification.

By clicking on *Run all* tests, the binary code will be downloaded to the target, the tests executed, and the result returned via the testIDEA *Test Status View* area.





# 7 CREATE A DERIVED TEST - TEST RESULTS



We can now review the test results in the test output view at the bottom of testIDEA.

The green background means that all the tests that have been executed have passed and our derived test is marked as OK

When the tests complete, testIDEA marks each test and section in the *Outline* window with a marker. If everything was OK, a green check mark is shown, otherwise we get a red mark with cross inside.

The screenshot displays the testIDEA interface with two main windows. The top window, titled 'Test Status', contains a table with the following data:

ID	Function/label	Message
---	C:\Users\bab...	All tests for selected editor completed successfully!Number of tests: 1/ : / [OK]

Below the table, a green button labeled 'CONNECTED' is visible. The bottom window shows the test output, which has a green background and contains the following text:

```
All tests for selected editor completed successfully!  
Number of tests: 1  
/:/  
[OK]
```

A yellow button labeled 'EVALUATION' is located at the bottom right of the output window.

# 7 CREATE A DERIVED TEST - META DATA



In order to use the test results for reporting purposes, we need to identify the tests executed. The *Meta* form can be used to give this test an identifier (ID).

- Test ID  
Test ID is used for documentation and maintaining a relationship to software requirements.
- Description  
Human readable **description** of the test.
- Tags  
Self-defined **tags** could be added, easing grouping of tests.

When finished click OK and the test appears in the Outline view.

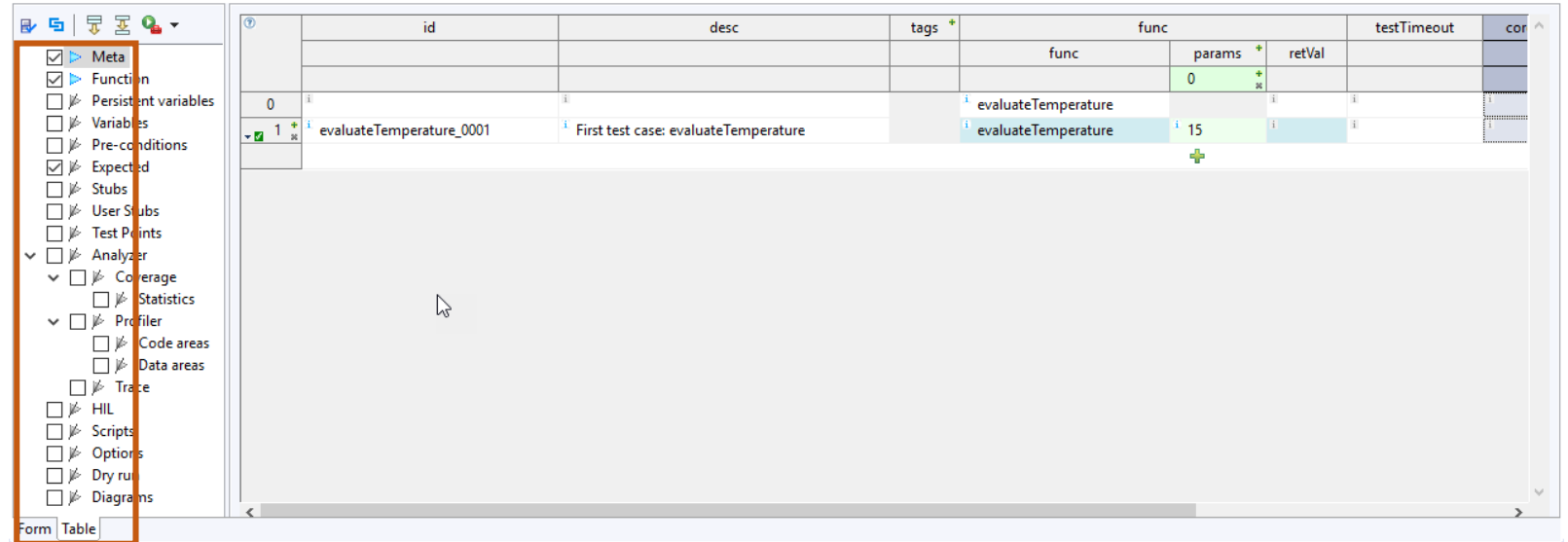
Congratulations! You have created your first test vectors!

## 8 ADDING MORE TESTS - TABLE OF TEST CASES

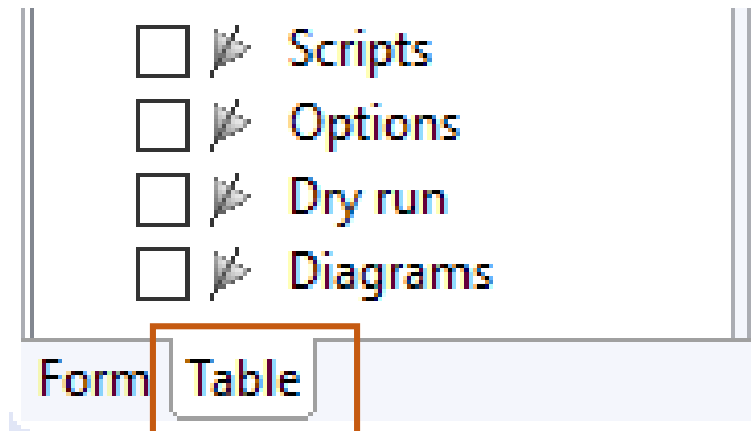


As we don't want to jump between different forms all the time when creating tests, it makes sense from this point on to use the table view to create further tests.

Changing to table view requires us to select the base test again in the *Outline*. Next, we have to choose some options from the test environment that we want displayed in the table. It is recommended to display at least Meta data, Function data and Expected return value data at this stage in our example.



	id	desc	tags	func	params	retVal	testTimeout	cor
0				evaluateTemperature	0			
1	evaluateTemperature_0001	First test case: evaluateTemperature		evaluateTemperature	15			



☐ Scripts

☐ Options

☐ Dry run

☐ Diagrams

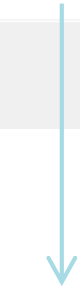
Form **Table**

If we now view the tests created, we can see the base test and our first derived test in a format not dissimilar to Excel.

A blue background on a field indicates an inherited, non-editable, element. This is the case in the example opposite for our first test, where the function name is inherited.

A green background is used for lists of values that can be passed as parameters. This is the case for the input parameter 15 that was entered in our first derived test.

	id	desc	tags	func			testTimeout	cor
				func	params	retVal		
0				evaluateTemperature	0			
1	evaluateTemperature_0001	First test case: evaluateTemperature		evaluateTemperature	15			



Blue:  
Inherited  
elements



Green:  
Individual elements that  
have already been entered  
in previous steps. These  
elements can be changed  
manually in the table.

This view allows us to quickly generate further tests very quickly based upon our base test.

To add another test simply click on the plus symbol and another test case will be created, with empty fields where test-dependent values can be entered.

	id	desc	tags	func			testTimeout	cor
				func	params	retVal		
0				evaluateTemperature	0			
1	evaluateTemperature_0001	First test case: evaluateTemperature		evaluateTemperature	15			
2				evaluateTemperature				

## 8 ADDING MORE TESTS - TABLE OF TEST CASES



Note: *It is not possible to copy the content of a cell and paste it into a cell as plain text.*

Reason: *When a cell is copied there is a lot of background information associated with the test copied with it, such as references to the original cell. If you attempt to paste this information into a cell as plain text, all the additional background information will also be inserted.*

*Try copying a cell and then pasting the data it into a text editor to better understand this issue.*

### Copy / Paste-options

1. Copy content of a cell and paste it into another selected cell
2. Copy text from a cell and paste it as text into a cell
3. Copy content of a whole test case option (such as *Expected values*) and paste it to another test case

## 8 ADDING MORE TESTS - TABLE OF TEST CASES



Using the boundary strategy as the basis for test creation, we can create all the test vectors discussed earlier for our *evaluateTemperature()* example from the base test.

The screen shot opposite shows how the resulting unit tests will appear with their unique *params* values and expected *expressions* results.

	func			testTimeout	coreId	assert		stackUs maxLin
	func	params	retVal			isExpectException	expressions	
		0					0	
0	evaluateTemperature							
1	evaluateTemperature	14					_isys_rv == TEMP_UNDER_15	
2	evaluateTemperature	15					_isys_rv == TEMP_OK	
3	evaluateTemperature	16					_isys_rv == TEMP_OK	
4	evaluateTemperature	39					_isys_rv == TEMP_OK	
5	evaluateTemperature	40					_isys_rv == TEMP_OK	
6	evaluateTemperature	41					_isys_rv == TEMP_OVER_40	

# 9 HANDLING TEST CASES



testIDEA offers a few further capabilities and concepts which may be helpful once basic test creation has been mastered. You can view these items in the following slides or select individual topics from the links on the right.

1. What to do when tests fail  
Go to [“When tests fail”](#)
2. Set test ID automatically  
Go to [“Set test ID automatically”](#)
3. Interpolation between parameters  
Go to [“Interpolation between parameters”](#)
4. Extrapolation between parameters  
Go to [“Extrapolation between parameters”](#)
5. Dry run mode  
Go to [“Dry run mode”](#)
6. Quick debug mode  
Go to [“Quick debug mode”](#)



# 9 HANDLING TEST CASES – WHEN TESTS FAIL



Here we review the example shown opposite:

During test creation, we wrongly expected the result “TEMP\_OVER\_40” when 40°C is passed in as the parameter to *evaluateTemperature()*.

When running the test, the following will be observed:

The background of the *Test Status* area turns red and statistics for the number of failed tests are displayed. Additionally, a list of exactly which of the tests failed is shown on the left hand side.

①	func			testTimeout	coreId	assert		stackUs	
	func	params +	retVal			isExpectException	expressions +		maxLin
		0 +					0 +		
0	i evaluateTemperature		i	i	i	=		i	
1 +	i evaluateTemperature	14	i	i	i	=	i _isys_rv == TEMP_UNDER_15	i	
2 +	i evaluateTemperature	15	i	i	i	=	i _isys_rv == TEMP_OK	i	
3 +	i evaluateTemperature	16	i	i	i	=	i _isys_rv == TEMP_OK	i	
4 +	i evaluateTemperature	39	i	i	i	=	i _isys_rv == TEMP_OK	i	
5 +	i evaluateTemperature	40	i	i	i	=	i _isys_rv == TEMP_OVER_40	i	
6 +	i evaluateTemperature	41	i	i	i	=	i _isys_rv == TEMP_OVER_40	i	
+									

Test Status			Message	Test report for selected editor, 6 test(s), 0 group(s): - 5 tests (83%) completed successfully - 1 test (17%) failed (invalid results)
ID	Function/la...			
---	C:\Users\ba...		Test report for selected editor, 6 test(s), 0 group(s):~ 5 tests (83%) complet...	
test2.9	evaluateTe...		Assert expression error:_isys_rv == TEMP_OVER_40 _isys_rv = 0x00000002...	
				test2.5 : / [OK] test2.6 : / [OK] test2.7 : / [OK] test2.8 : / [OK] test2.9 : / [FAILED] test2.10 : / [OK]

# 9 HANDLING TEST CASES – WHEN TESTS FAIL



A left mouse click on the failed test provides you with more information. In the provided example the test failed due to an *Assert expression error*.

We expected “TEMP\_OVER\_40” and we actually were returned the value 2 during testing which equates to the value “TEMP\_OK” as defined in the enumeration. Such information helps us to pinpoint the source of the issue.

Now it is up to the tester to determine if the test was incorrectly created (perhaps because the specification for the function was misunderstood), or if the function is incorrectly programmed. If it is the latter, a bug report can be submitted.

ID	Function/la...	Message
---	C:\Users\ba...	Test report for selected editor, 6 test(s), 0 group(s):- 5 tests (83%) complet...
test2.9	evaluateTe...	Assert expression error: _isys_rv == TEMP_OVER_40 _isys_rv = 0x00000002...

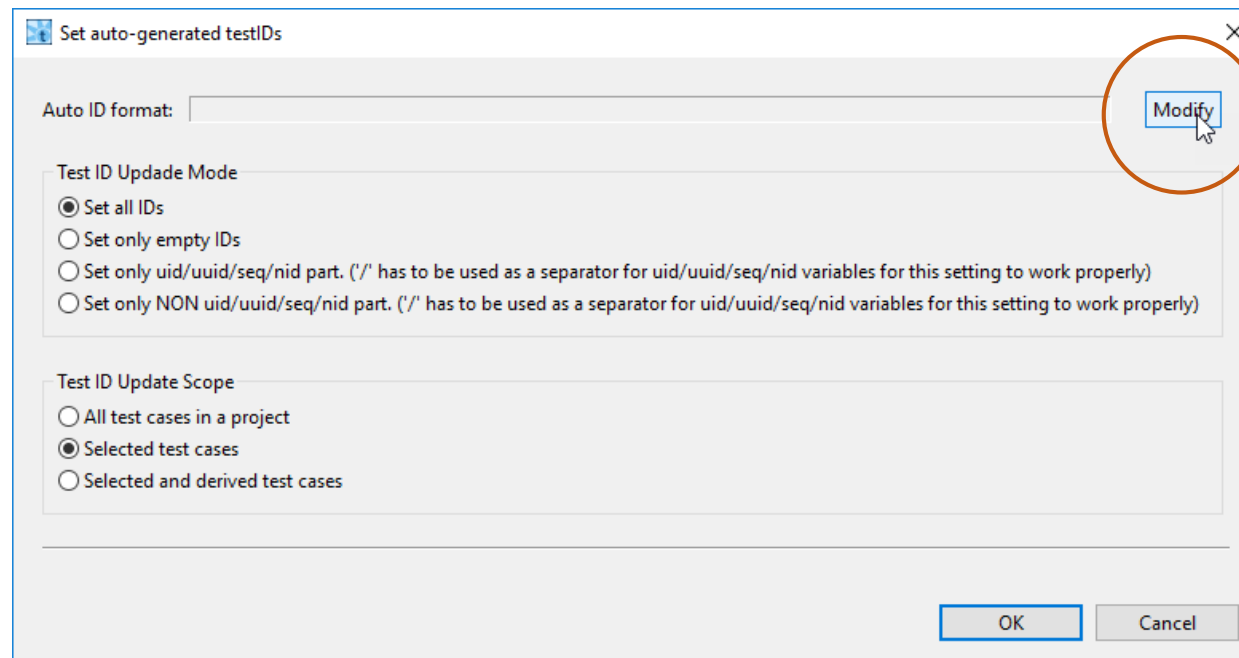
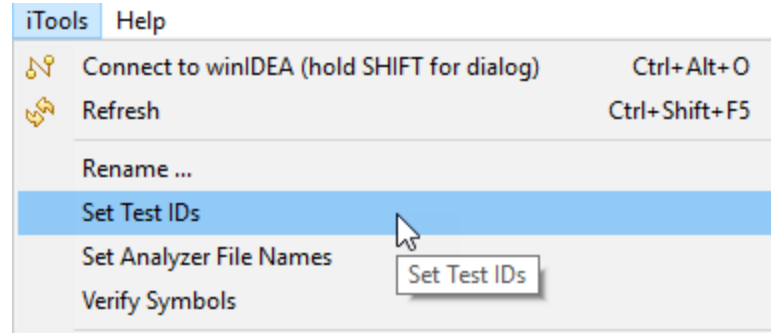
```
Assert expression error:  
_isys_rv == TEMP_OVER_40  
_isys_rv = 0x00000002 (2)  
TEMP_OVER_40 = 0x03 (3)  
{}
```

# 9 HANDLING TEST CASES – SET TEST CASE ID AUTOMATICALLY



Manually tagging each test with a unique ID can be automated if preferred.

In the *iTools* menu the option “Set Test IDs” can be found. This opens a new dialogue with different options for configuring automatically generated test IDs. The format of the Auto-ID can be defined via the “Modify” button.



# 9 HANDLING TEST CASES – SET TEST CASE ID AUTOMATICALLY

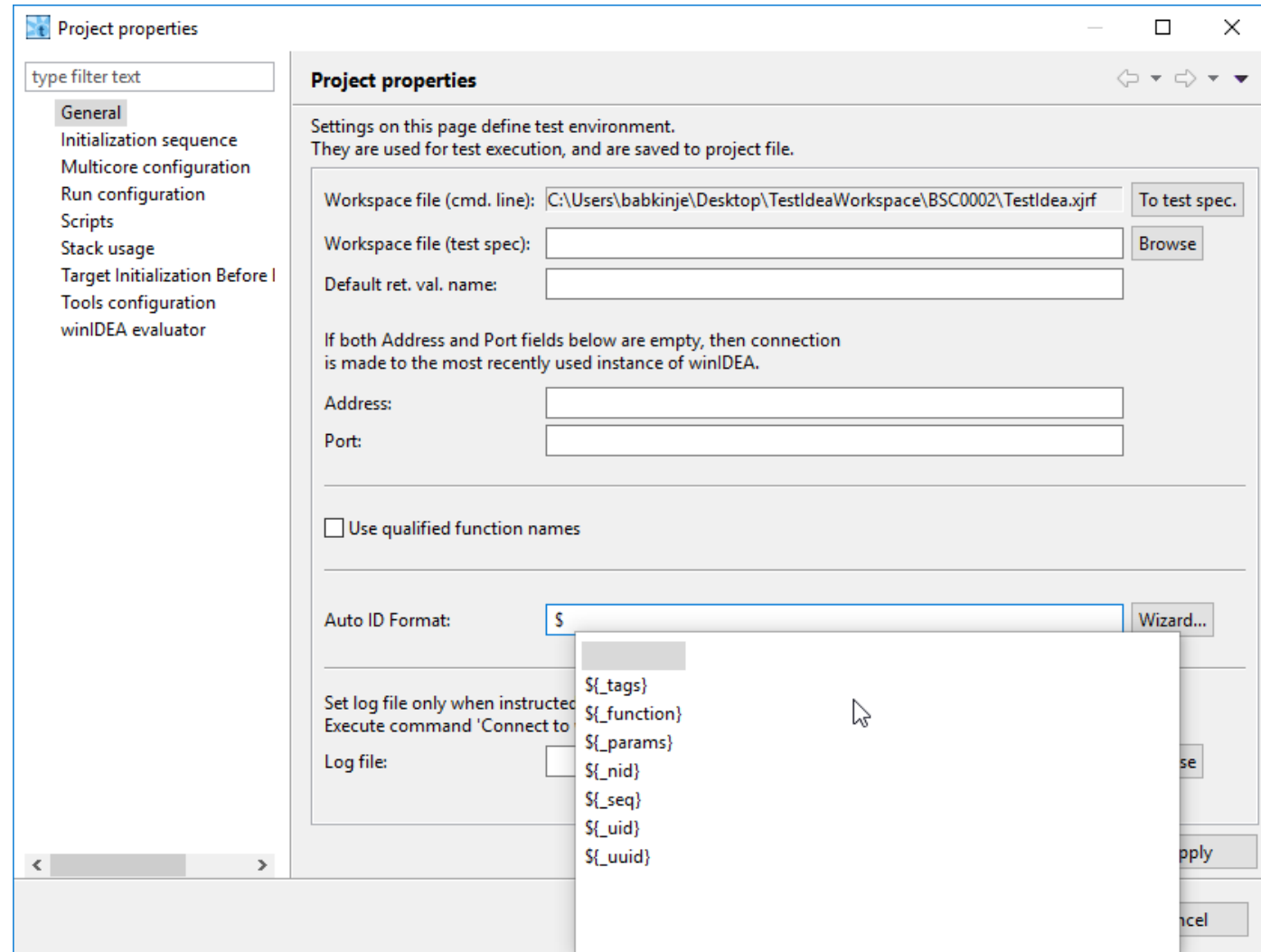


Modify opens “*Project Properties*” in the “*General*” section.

Here the *Auto ID Format* can be configured. The ID can be formed from a combination of fixed text (e.g. “Project\_Randle\_”) and elements that can be referenced from the source code or the test. For example, *\$\_function* will insert the function name into the Test ID, while *\$\_params* will insert the test’s parameters.

Numerical IDs, such as sequential numbers ( *\$\_seq* ) can also be generated automatically. Finally, entries entered into the “Tag” field of a test can be inserted using the formatter *\$\_tags*. This can be useful for grouping test results together.

The *Wizard...* button provides further options.



# 9 HANDLING TEST CASES – INTERPOLATION



To fill data between test cases with interpolated values simply provide empty fields between the start and end value, mark the empty fields and the fields containing the border values, and click the button “Interpolate between first and last cell in selected region of table column”

The empty fields will then be filled with the interpolated values.

	func	testTimeout	coreId	isExpectException	assert	stackUsage
0						
1	0				<i>i</i> _isys_rv == TEMP_UNDER_15	
2	30				<i>i</i> _isys_rv == TEMP_OK	
3	60				<i>i</i> _isys_rv == TEMP_OVER_40	
4	-100				<i>i</i> _isys_rv == TEMP_UNDER_15	
5						
6						
7	45				<i>i</i> _isys_rv == TEMP_OVER_40	

The interpolated values now have been filled in.

**Note:** at the time of writing, numbers were truncated in testIDEA (up to and including version 9.17.25). Future versions of testIDEA will round down for values < 0.5 and round up for values >= 0.5.

	func	testTimeout	coreId	assert	stackUsage
				isExpectException	expressions
	0				0
0					
1	0				<code>_isys_rv == TEMP_UNDER_15</code>
2	30				<code>_isys_rv == TEMP_OK</code>
3	60				<code>_isys_rv == TEMP_OVER_40</code>
4	-100				<code>_isys_rv == TEMP_UNDER_15</code>
5	-52				
6	-4				
7	45				<code>_isys_rv == TEMP_OVER_40</code>

Click “*Extrapolate*” and the selected region of table column will be filled with extrapolated values.

Extrapolate first two cells in selected region of table column.

	func	testTimeout	coreId	isExpectException	expressions	stackUsage
<input checked="" type="checkbox"/>	Function	0			0	
<input type="checkbox"/>	Persistent vari	0				
<input type="checkbox"/>	Variables	1			<input checked="" type="checkbox"/> _isys_rv == TEMP_UNDER_15	
<input type="checkbox"/>	Pre-condition	2			<input checked="" type="checkbox"/> _isys_rv == TEMP_OK	
<input checked="" type="checkbox"/>	Expected	3			<input checked="" type="checkbox"/> _isys_rv == TEMP_OVER_40	
<input type="checkbox"/>	Stubs	4			<input checked="" type="checkbox"/> _isys_rv == TEMP_UNDER_15	
<input type="checkbox"/>	User Stubs	5			<input checked="" type="checkbox"/> _isys_rv == TEMP_UNDER_15	
<input type="checkbox"/>	Test Points	6				
<input checked="" type="checkbox"/>	Analyzer	7				
<input type="checkbox"/>	Coverage	8				
<input type="checkbox"/>	Statisti	9				
<input checked="" type="checkbox"/>	Profiler					
<input type="checkbox"/>	Code a					
<input type="checkbox"/>	Data ar					
<input type="checkbox"/>	Trace					
<input type="checkbox"/>	HIL					
<input type="checkbox"/>	Scripts					
<input type="checkbox"/>	Options					
<input type="checkbox"/>	Dry run					

Form Table

# 9 HANDLING TEST CASES - EXTRAPOLATION



The extrapolated values have now been filled in.

	func		testTimeout	coreId	isExpectException	assert	stackUsage
	params	retVal				expressions	maxLimit
	0					0	
0							
1	0					<i>i</i> _isys_rv == TEMP_UNDER_15	
2	30					<i>i</i> _isys_rv == TEMP_OK	
3	60					<i>i</i> _isys_rv == TEMP_OVER_40	
4	-5					<i>i</i> _isys_rv == TEMP_UNDER_15	
5	5					<i>i</i> _isys_rv == TEMP_UNDER_15	
6	15						
7	25						
8	35						
9	45						



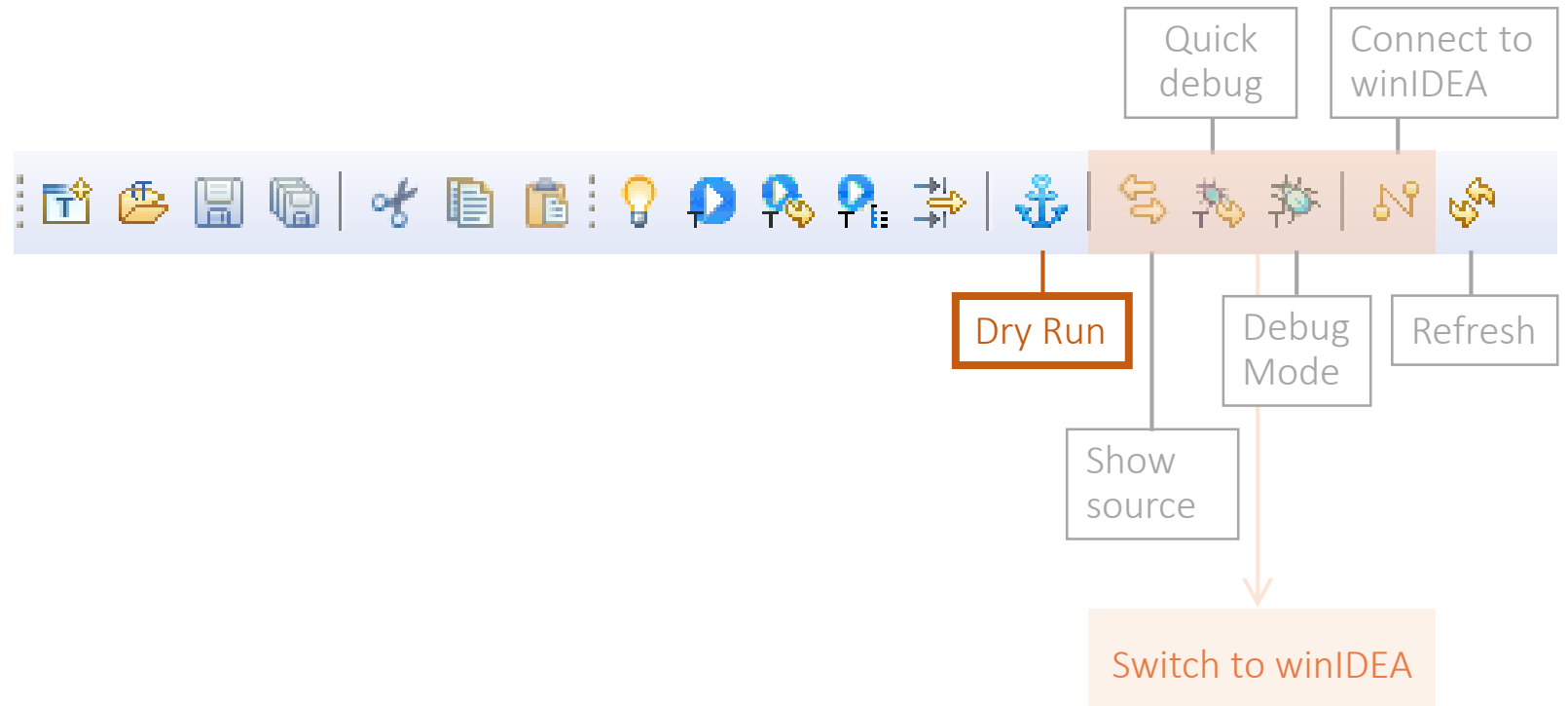
# 9 HANDLING TEST CASES – DRY RUN MODE



## Dry run mode

This functionality can be used to record outcome of existing tests before we modify our source code. With the test case generator, create a set of test cases and then use dry run to record the state of the test and analyzer results for each test case.

After modifying the target code and rerunning the tests, the test results can show us what has changed.



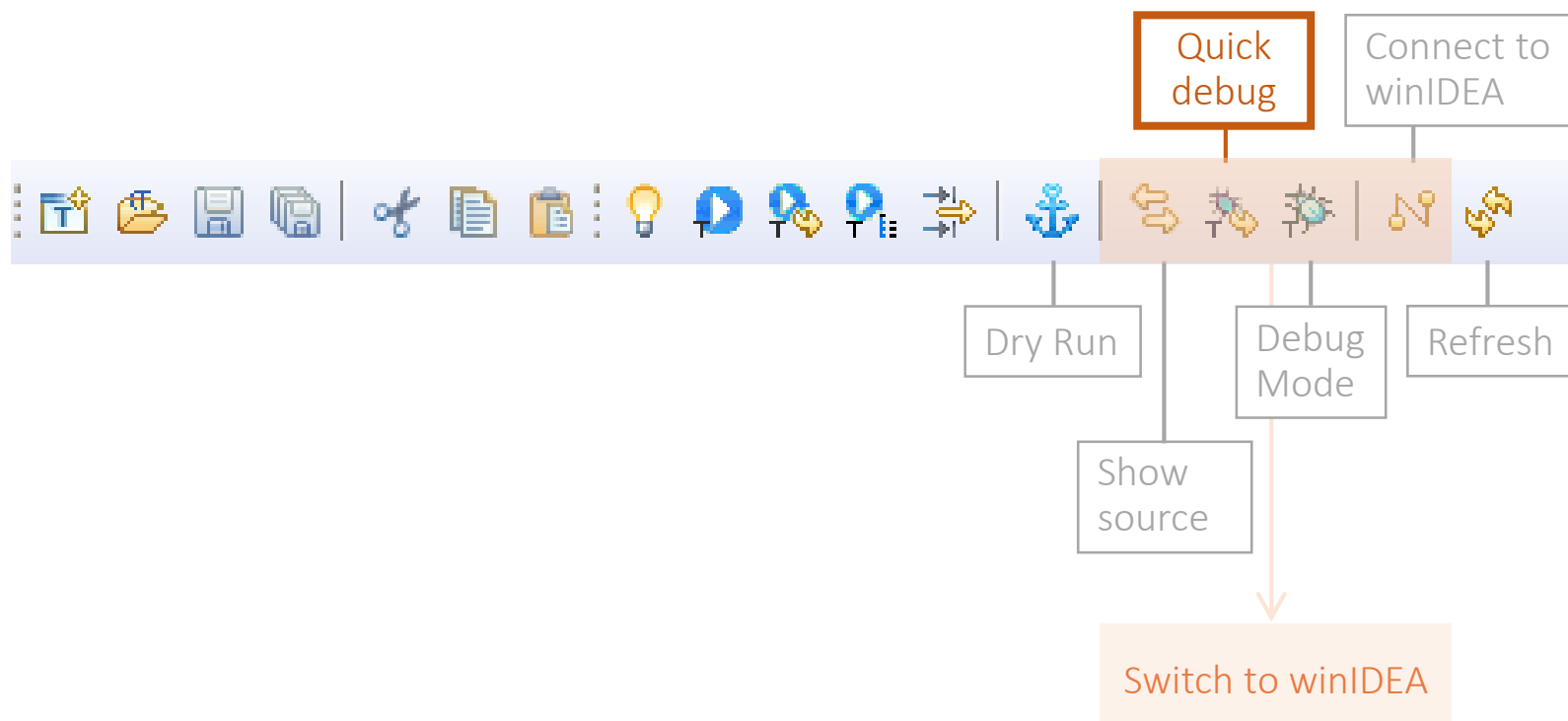
# 9 HANDLING TEST CASES – QUICK DEBUG MODE

## Main tool bar – Debug options

Generally speaking, these options enable the test developer to switch to the winIDEA environment during test execution on the target, enabling use of debug features or to analyze functionality related to the source code itself.

## Quick debug

This functionality runs the selected test on the target but stops execution at the function entry point. The test developer can then execute the function as desired (using breakpoints, stepping, etc.) until completion. Upon reaching the end of the function, the testIDEA environment is re-engaged.



## SUMMARY

---

# testIDEA

- Start with a **non-executable base** test which includes the minimum required information that is **common to all tests**
- For further **derived tests**, the core elements of the base tests are inherited. All further unique parameters, such as the test's input parameter(s) and the expected response value(s) have to be **filled in individually**
- The table view helps to quickly create test cases in order to get a series of test vectors that provide us with the desired test coverage

