During testing

STUBBING HARDWARE

Objectives

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

- Explain the use cases for stubbing during unit tests
- List two alternative ways to stub hardware functionality using testIDEA
- Create a stub within testIDEA

testIDEA

Contents

STUBBING HARDWARE

testIDEA

1 WORKING WITH STUBS

When developing unit tests, it may be the case that a function being tested is dependent upon:

- Hardware that is not yet available.
- Software that has not yet been written.
- An element that provides non-deterministic data (e.g. a sensor).

In such cases, it is possible to 'stub' the section of affected code for the purposes of testing.

This essentially means:

- Replacing the function call with the injection of the data the caller would have returned.
- Linking the function call to a function especially written for testing purposes.

```
bool setDelay(void) {
    bool returnValue = true;
    unsigned int adcValue = 0;

    adcValue = getADC();

    if (adcValue >= 512) {
        returnValue = false;
    }

    return returnValue;
}
```

```
int getADC(void) {
    ...
    return adcValue;
}
```

1 WORKING WITH STUBS

testIDEA can, solely for the purposes of testing, be configured to artificially call a C/C++ function that has been written especially for testing.

The function to be used will need to be available in the list of symbols of the ELF-format file being testing. This function will typically be written to return deterministic values for a sensor, but could be programmed to perform any suitable task.

```
boolean setDelay(void) {
    boolean returnValue = true;
    unsigned int adcValue = 0;

    adcValue = getADC();

    if (adcValue >= 512) {
        returnValue = false;
    }

    return returnValue;
}
```

```
int getADC(void) {
    ...
    return adcValue;
}
```

```
int myGetADC(void) {
    ...
    return myAdcValue;
}
```

1 WORKING WITH STUBS

Alternatively, testIDEA can be configured to insert values from a table into the caller's variable.

In the example shown here, rather than calling the function *getADC()*, testIDEA simply inserts values in a predetermined order into the caller's storage variable *adcValue* at each execution of the call.

In this Unit we will be focusing on the stubbing method shown here on this slide.

It is important to remember that using stubs will change the execution time of the code. This should not be an issue in the context of unit testing.

```
boolean setDelay(void) {
    boolean returnValue = true;
    unsigned int adcValue = 0;

    adcValue = getADC();

    if (adcValue >= 512) {
        returnValue = false;
    }

    return returnValue;
}
```

stubs		
0		
stubbedFunc	retValName	assignSteps
		0
		assign
		adcValue
getADC	adcValue	
getADC	adcValue	0
getADC	adcValue	1
getADC	adcValue	511
getADC	adcValue	512
getADC	adcValue	513

1 WORKING WITH STUBS

In the example shown here the function *setDelay()* is directly dependent on the result from a call to the hardware dependent function *getADC()*.

Good abstraction of code would normally see the result from the ADC measurement being passed into the *setDelay()* function, rather than acquiring the value through a function call in the function itself. However, for the purposes of demonstration, this example code is easy to follow.

```
unsigned int getADC(void) {
    unsigned int returnValue = 0;

    returnValue = analogRead(3);

    return returnValue;
}

bool setDelay(void) {
    bool returnValue = true;
    unsigned int adcValue = 0;

    adcValue = getADC();

    if (adcValue >= 512) {
        returnValue = false;
    }

    return returnValue;
}
```

1 WORKING WITH STUBS

The *setDelay()* function returns false when the ADC returns a value of 512 or greater. In order to test this functionality we will need to force the tests to insert suitable, deterministic values into the variable *adcValue* instead of calling the hardware-dependent function *getADC()*.

Using the boundary method seen before, it would be recommended to test values of 0, 1, 511, 512 and 513, with expected responses being as shown in the table opposite.

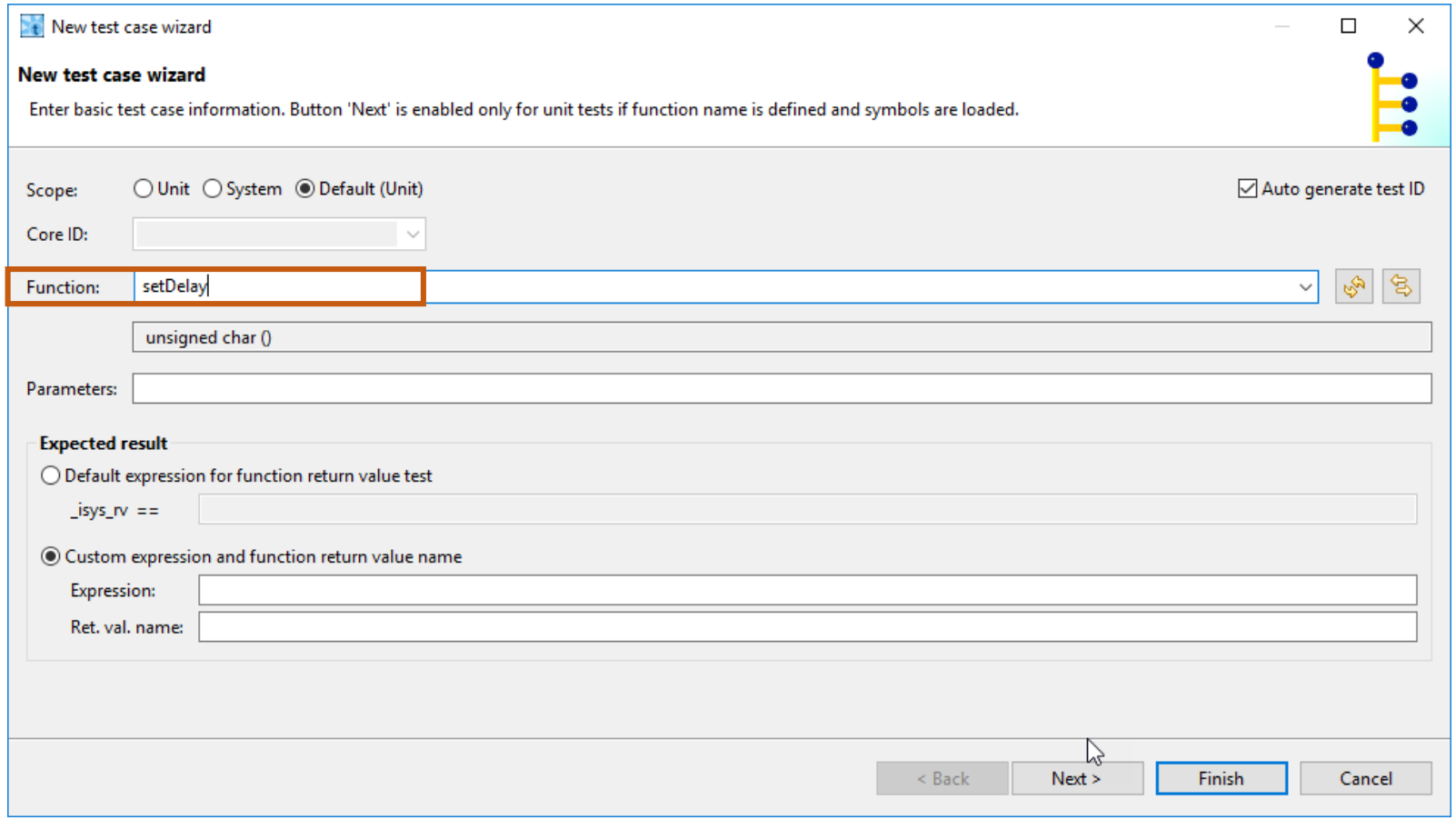
```
unsigned int __attribute__ ((noinline)) getADC(void) {
    unsigned int returnValue = 0;

    returnValue = analogRead(3);

    return returnValue;
}

boolean __attribute__ ((noinline)) setDelay(void) {
    boolean returnValue = true;
    unsigned int adcValue = 0;

    adcValue = getADC();

    if (adcValue >= 512) {
        returnValue = false;
    }

    return returnValue;
}
```

Input	Expected Response
0	True
1	True
511	True
512	False
513	False

2 CREATE BASE TEST

As seen previously, we start by creating a non-executable base class.

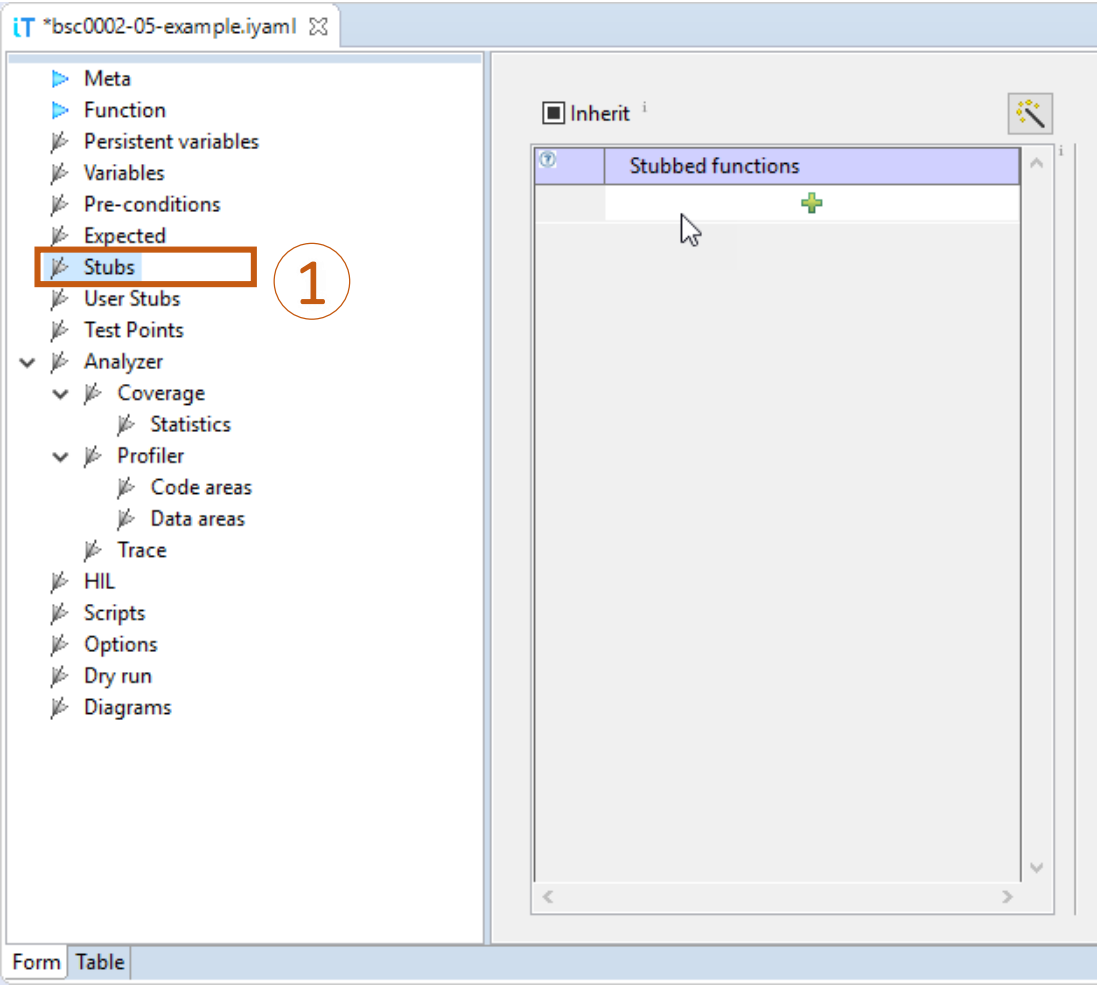
The *setDelay()* function will then be tested by for a pass/fail outcome by comparing the return value with 1 or 0 (true and false). As this is test-dependent, we will leave the *Expected result* field in the base test empty, as is shown here.



3 SET STUB IN BASE TEST

After choosing the function we have to create a Stub in the Base Test.

1 Switch to the *Stubs* form.

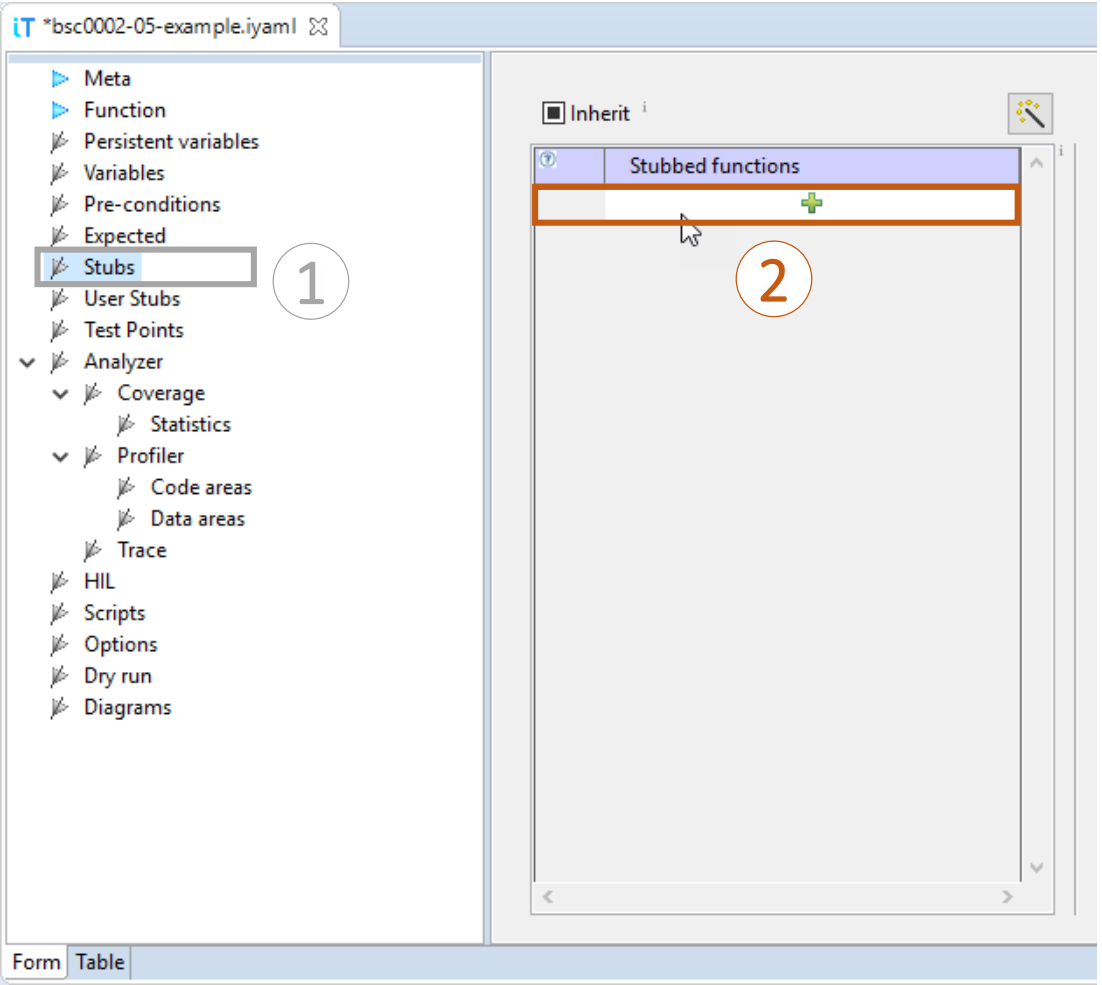


3 SET STUB IN BASE TEST

After choosing the function we have to set a Stub in the Base Test.

1 Switch to the *Stubs* form.

2 Next, add the stubbed function by clicking the plus symbol as shown.



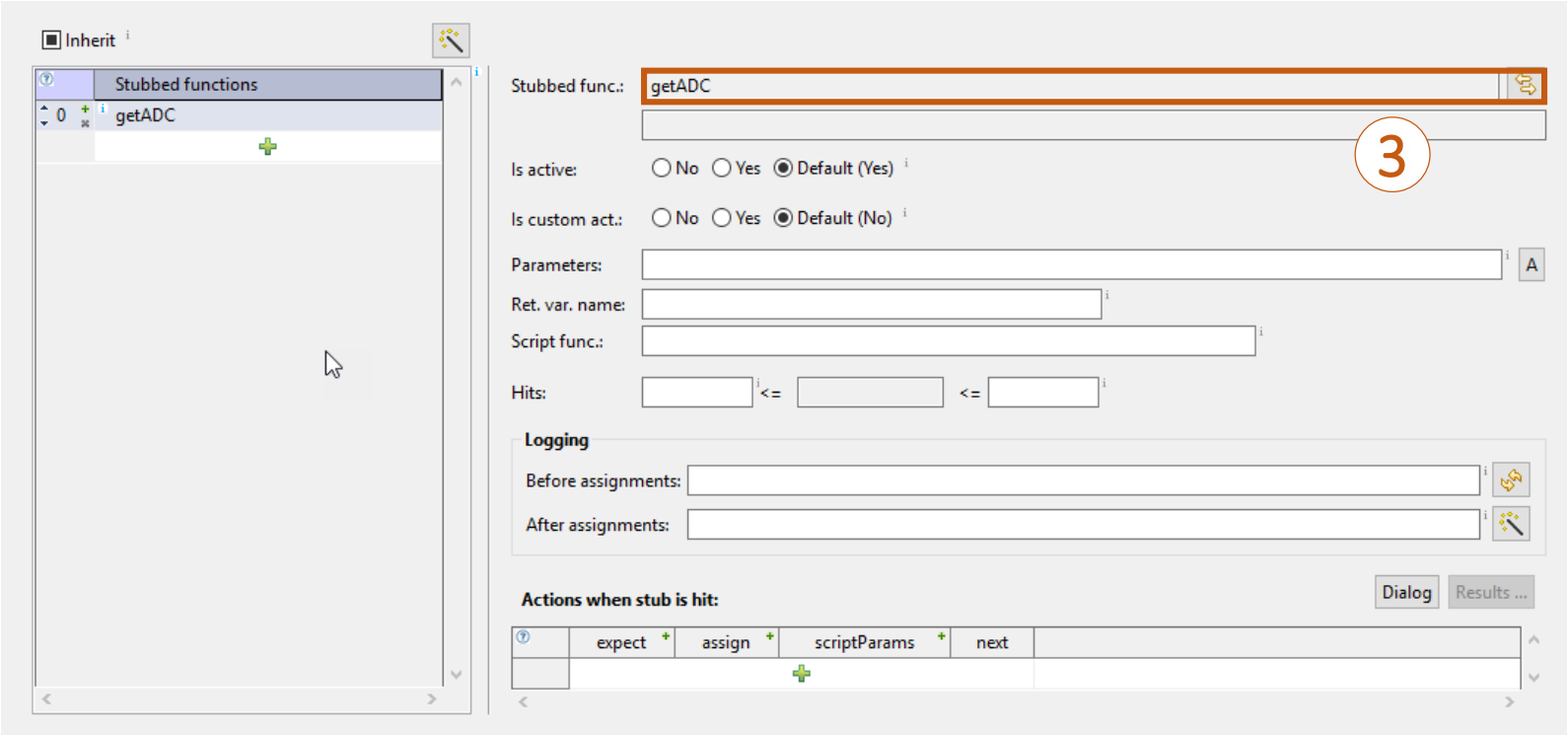
3 SET STUB IN BASE TEST

After choosing the function we have to set a Stub in the Base Test.

1. Switch to the *Stubs* form.

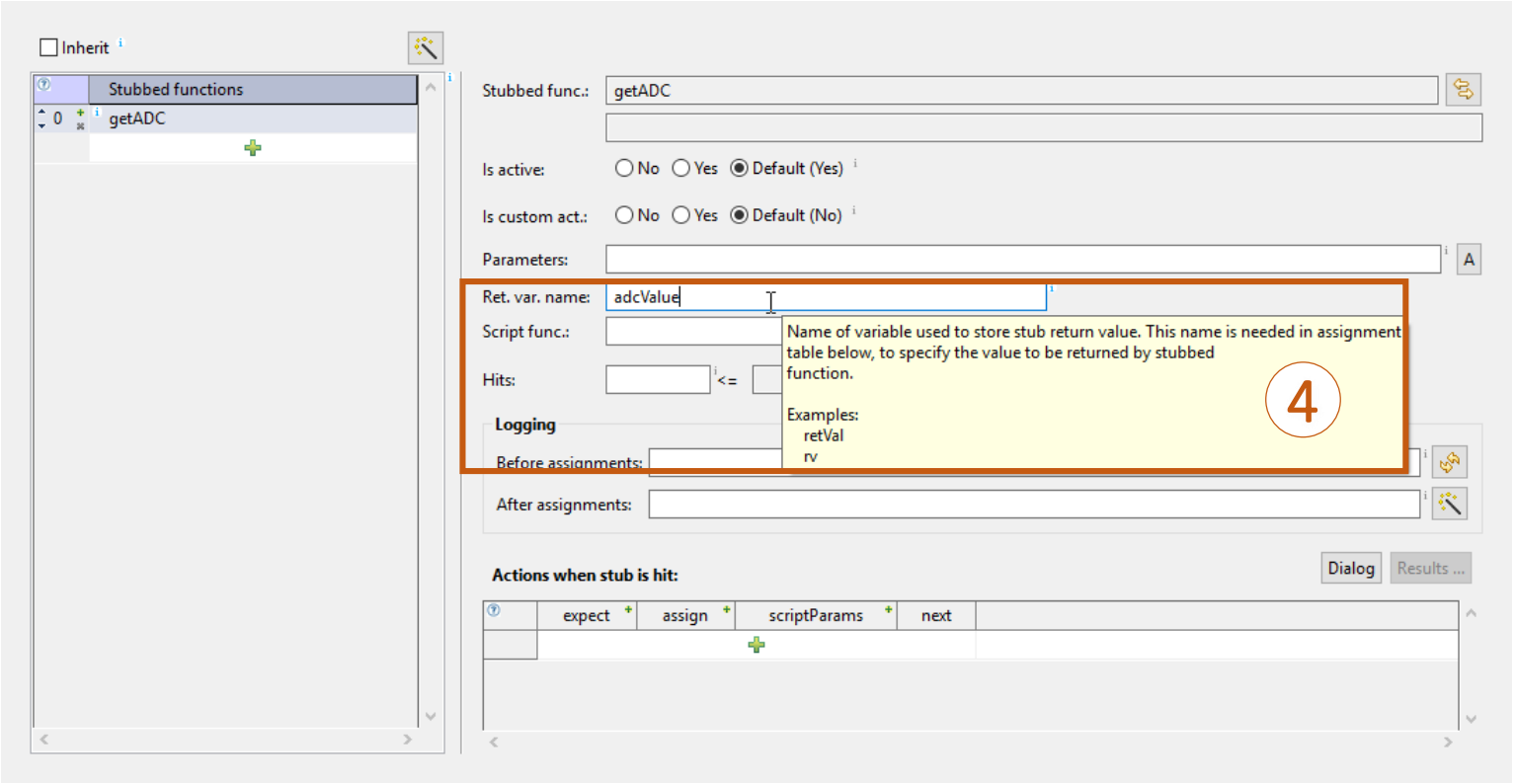
2. Next, add the stubbed function by clicking the plus symbol as shown.

3. The function *getADC()* is then added as the function to be stubbed.



3 SET STUB IN BASE TEST

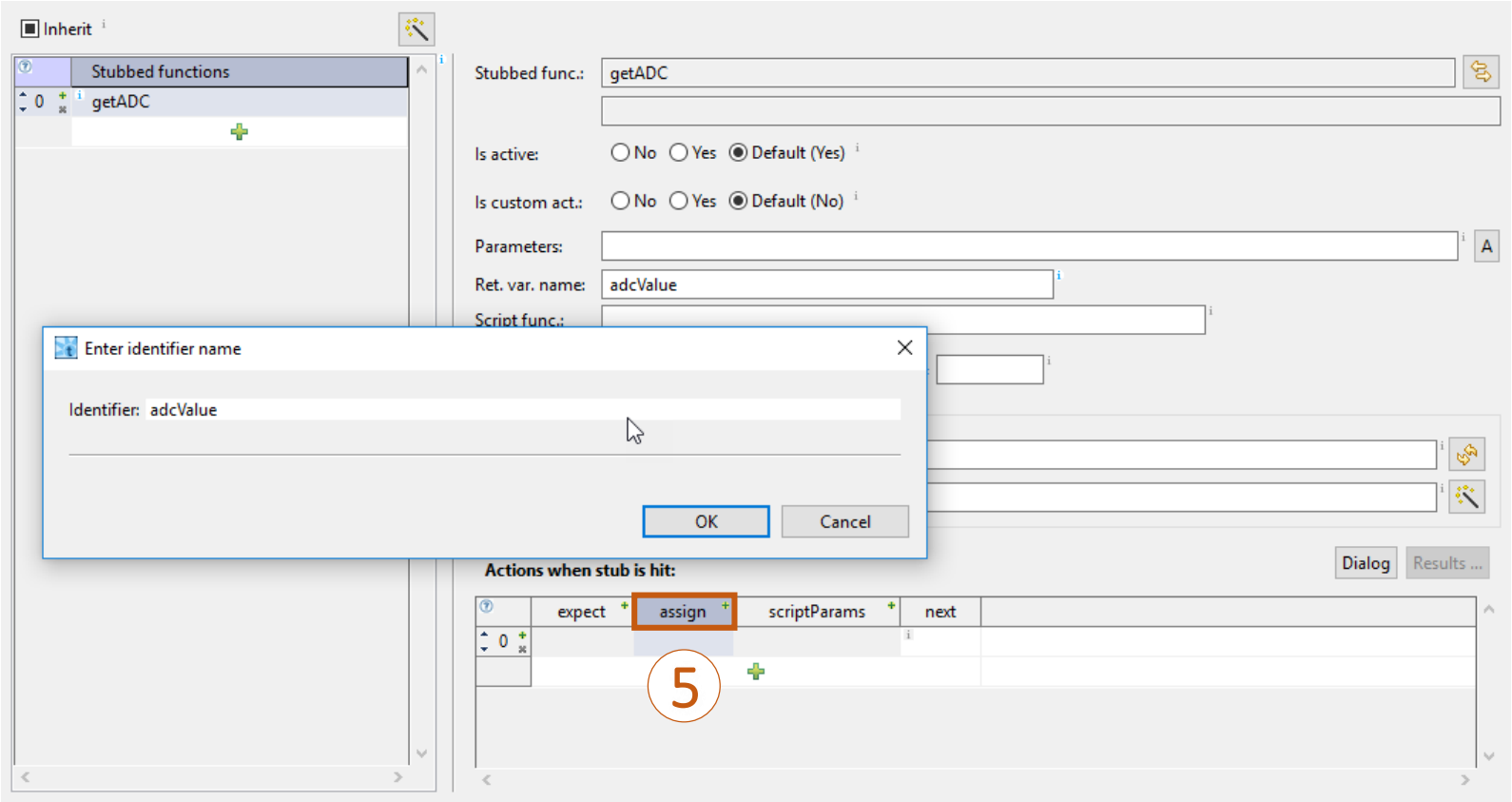
4 As the *Return variable name*, *adcValue* is entered. This is the variable we want to overwrite using the stub functionality.



3 SET STUB IN BASE TEST

5 Each time the stub function is activated the defined action should be triggered.
Select the *Plus* symbol in the *assign* column and select the variable which should be overwritten by the stub – in this case *adcValue*.

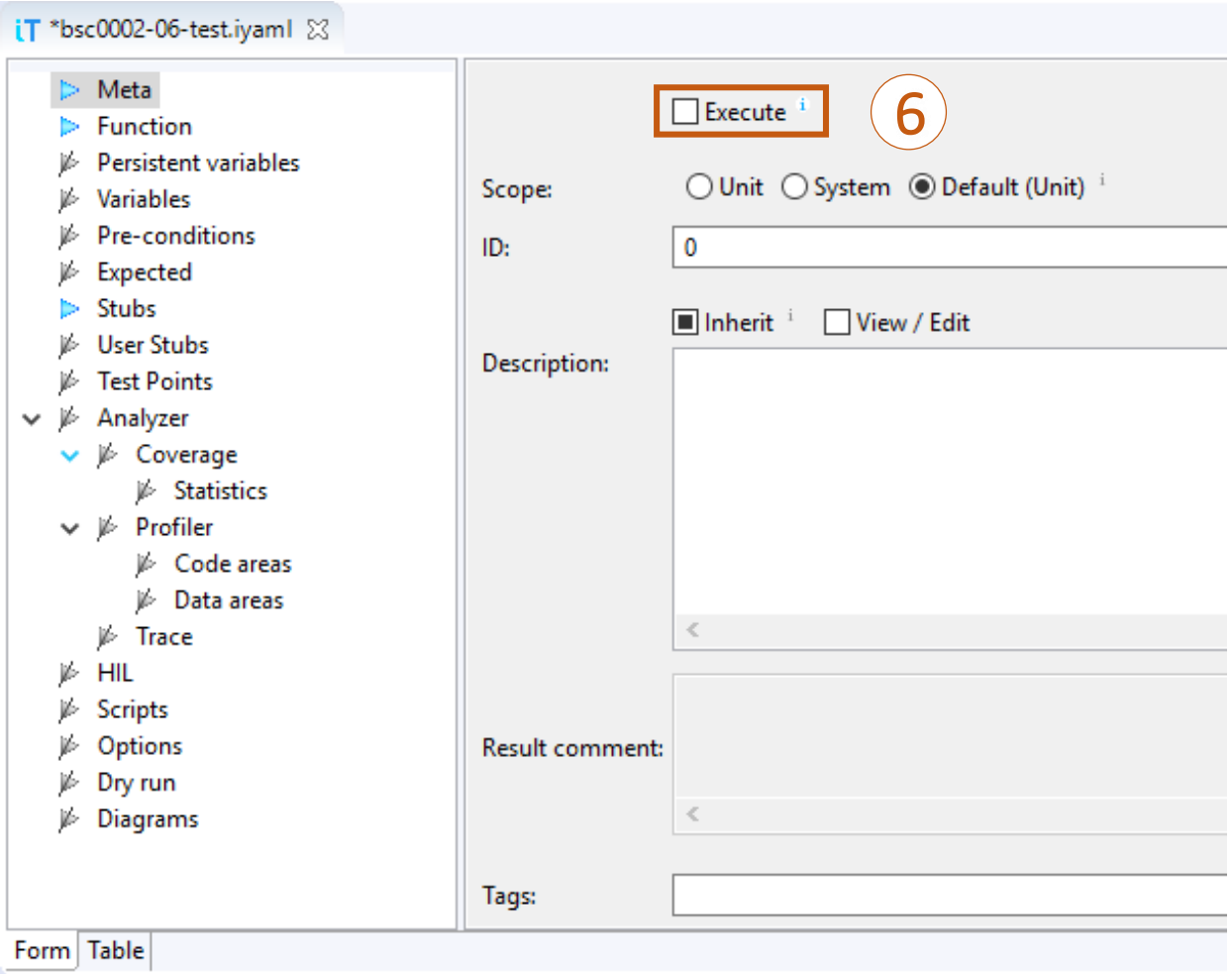
The actual value to be inserted will be defined in each derived test.



3 SET STUB IN BASE TEST

6 Finally, in the *Meta* form, uncheck the *Execute* option.

The base test is now complete and we can continue by creating derived tests.



4 CREATE DERIVED TESTS

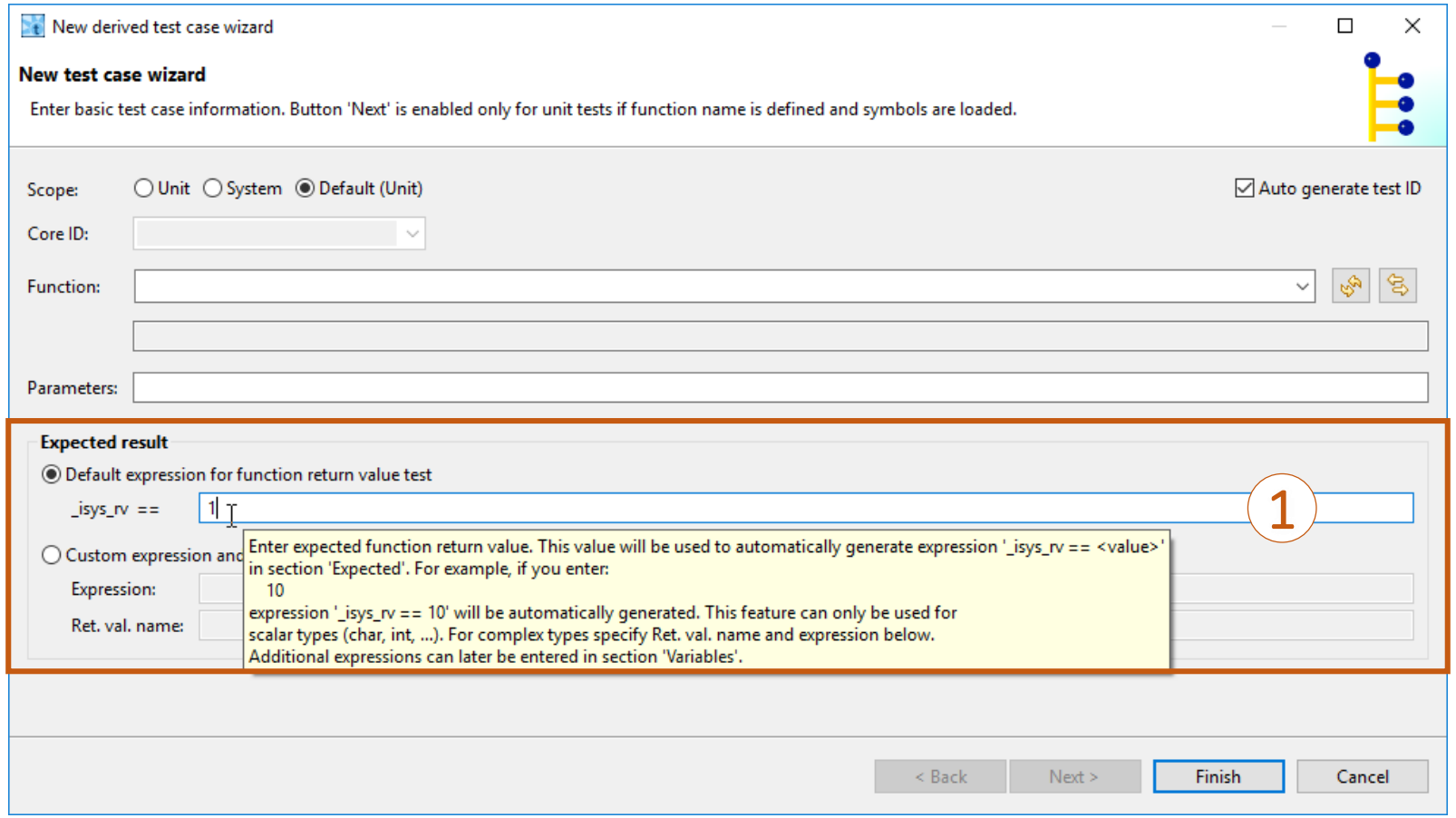
Create a new derived test

The derived test inherits almost everything from the base test. Only the *Expected result* and the *Stub* need to be configured.

For our first executable test the value ‘0’ will be returned from the stubbed function, resulting in the *setDelay()* function returning ‘1’ or true.

1 After selecting the base test and selecting *Test -> New Derived Test...* the *Expected result* value can be defined in the wizard as shown here.

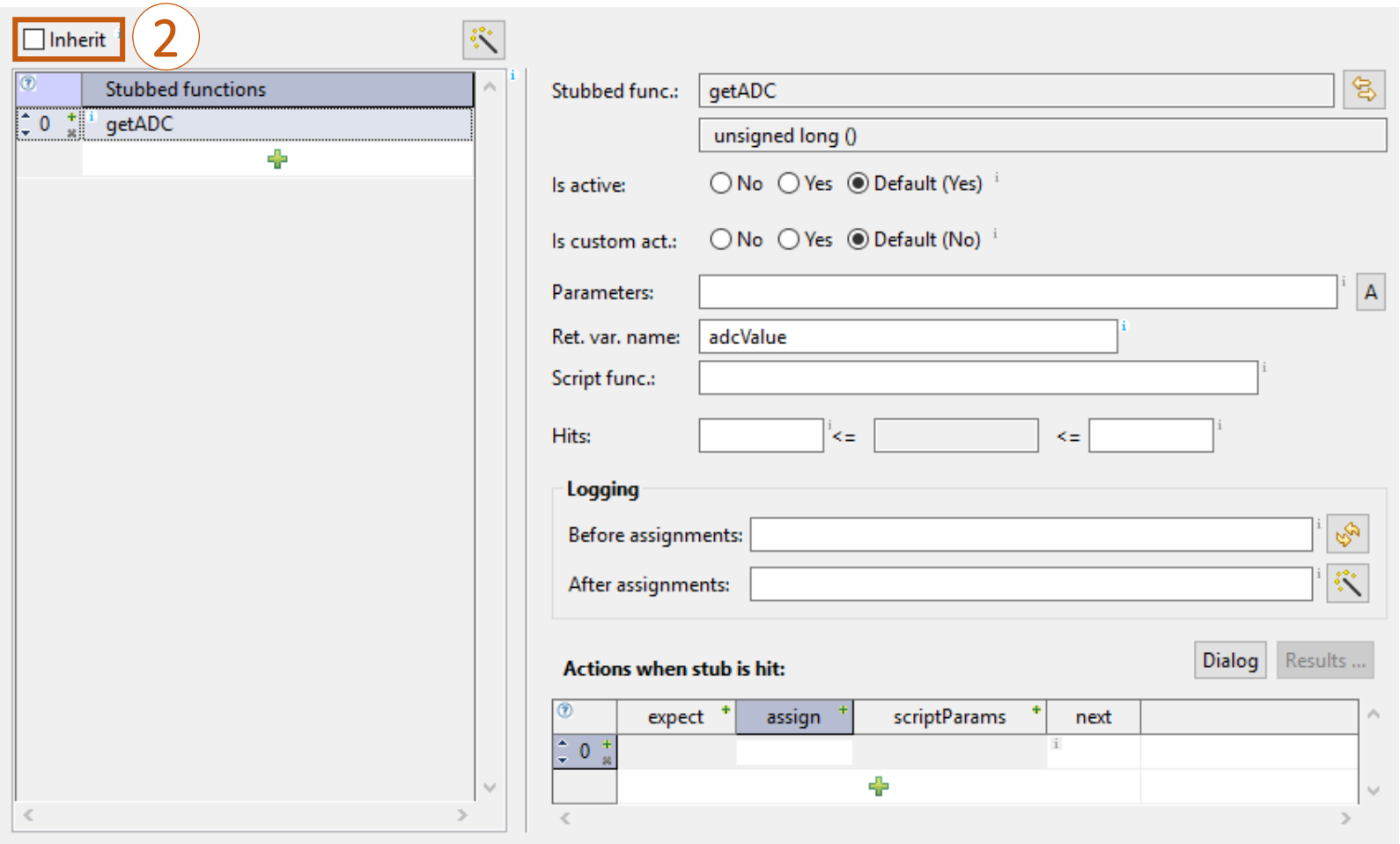
Once complete, the *Finish* button is clicked to close the dialog.



4 CREATE DERIVED TESTS

Now we have to add the information for the stub for this test. Select the *Stub* form again

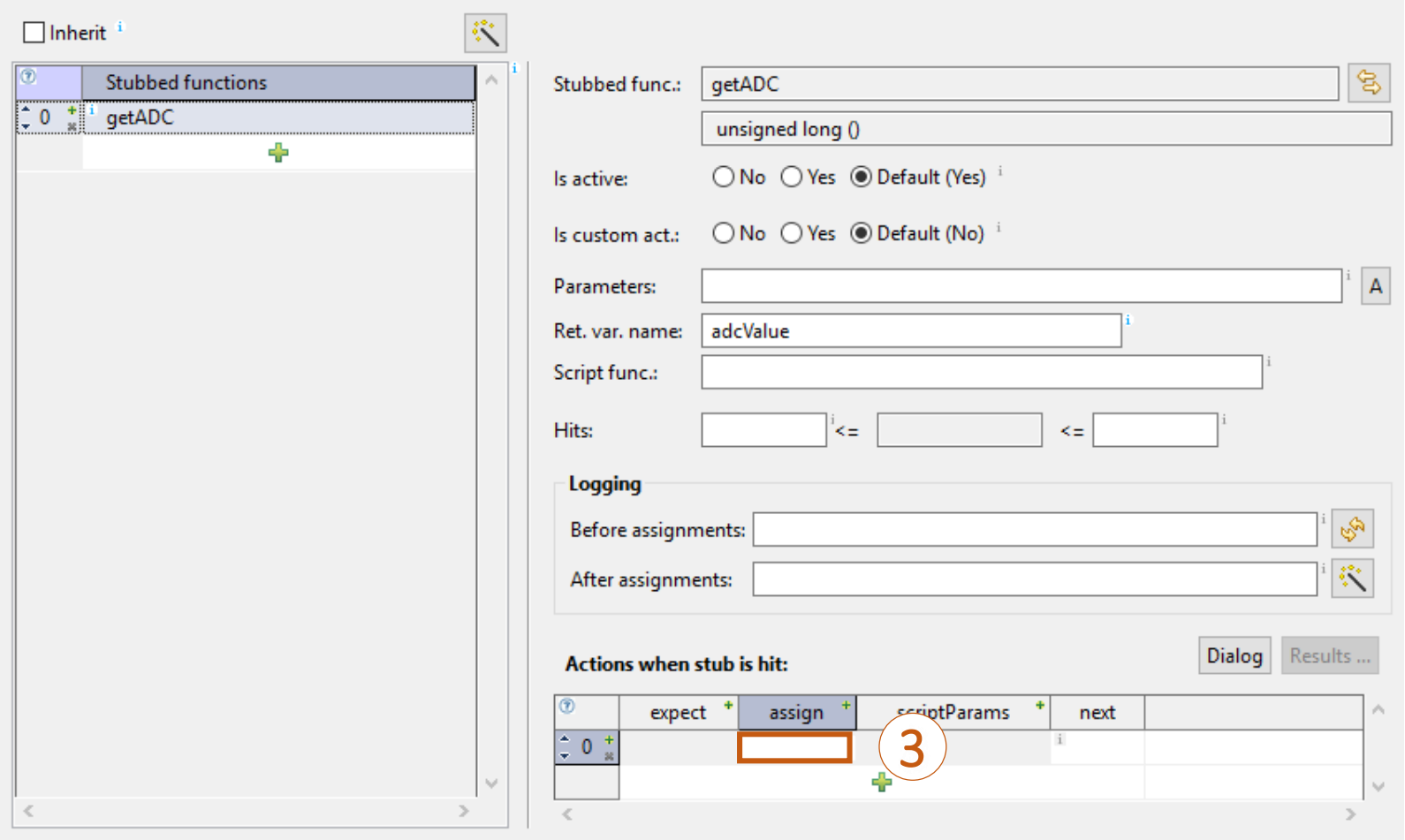
2 Click the *Inherit* check-box twice to change the status to *unchecked*.
The input settings remain as defined but the fields are now editable.



4 CREATE DERIVED TESTS

3 Set the stub value:

Now the desired value for *adcValue* has to be entered for this test.



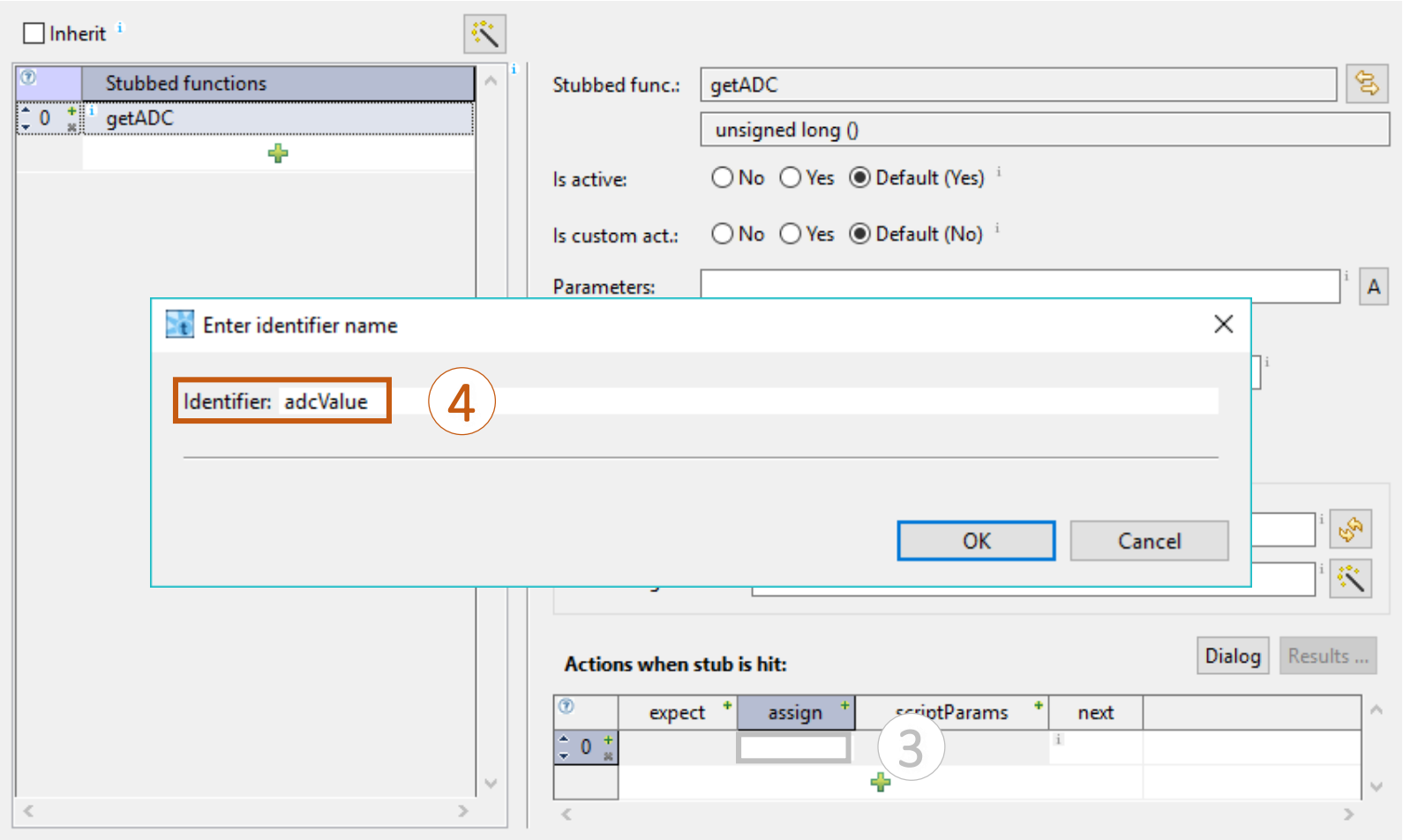
4 CREATE DERIVED TESTS

3 Set the stub value:

Now the desired value for *adcValue* has to be entered for this test.

4 Finally, enter the name of variable to be changed.

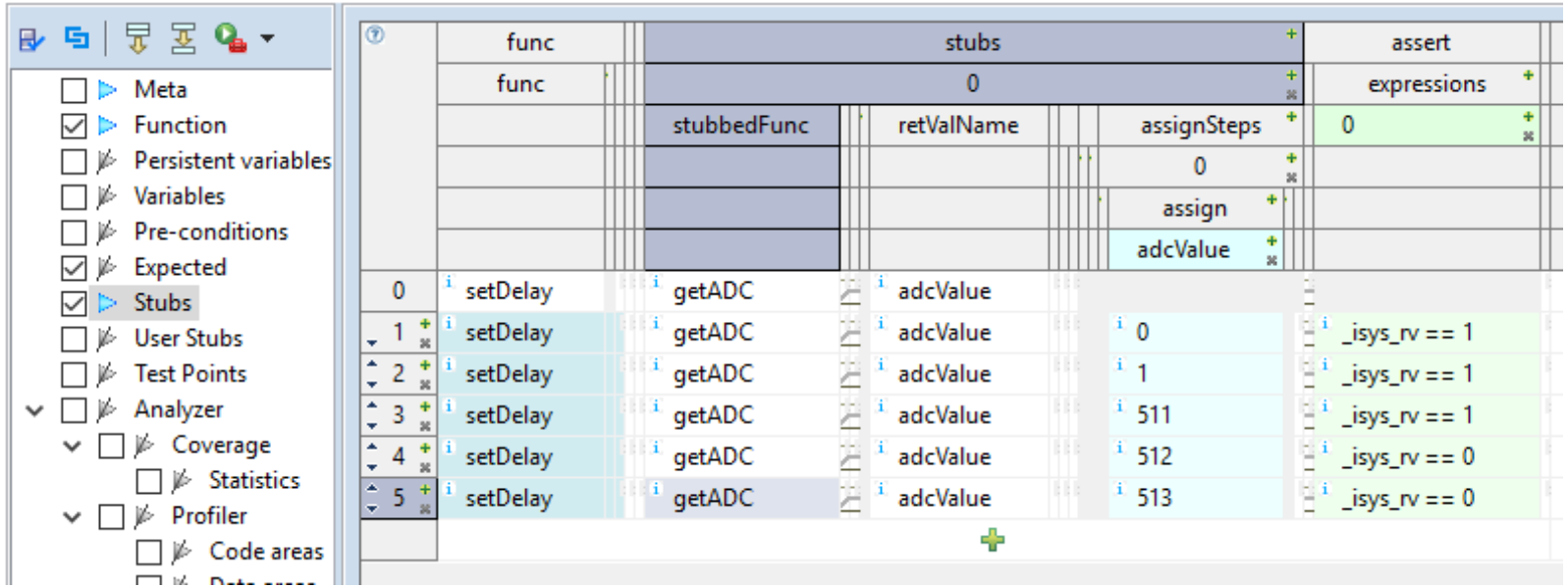
Upon completion of this first executable test, the remaining tests can be easily modified using the table view.



5 TABLE VIEW

By switching to *Table* view, further test vectors can be easily created.

By utilizing boundary testing method, the final collection of test vectors should look similar to that shown here.



SUMMARY

testIDEA

6 SUMMARY

- To ease or simplify testing, functions that are hardware dependent or are not yet written can be stubbed.
- Stubbing allows deterministic data to be inserted into a variable for the purposes of testing.
- Alternatively, function calls could be linked to alternate function written in C/C++.