ASSA ABLOY

Unit Test Tutorial

An introduction and tutorial for the icTest unit test system

# Table of Contents

[1 Table of Contents 2](#_Toc366477234)

[2 Revision History 4](#_Toc366477235)

[3 Introduction 4](#_Toc366477236)

[3.1 Purpose 4](#_Toc366477237)

[3.2 Scope 4](#_Toc366477238)

[3.3 Definitions and abbreviations 4](#_Toc366477239)

[3.4 References 5](#_Toc366477240)

[4 Motivation for Unit Tests and Test Driven Development 6](#_Toc366477241)

[5 Unit Test Example 6](#_Toc366477242)

[6 A Walkthrough of the Files 8](#_Toc366477243)

[6.1 Tests/nnnIcTest.c 8](#_Toc366477244)

[6.2 Tests/module.mk 9](#_Toc366477245)

[6.3 TestDouble / Fake 9](#_Toc366477246)

[6.4 Running the Test 10](#_Toc366477247)

[6.4.1 Test Assertions 10](#_Toc366477248)

[6.4.2 Handling (Testing) Error Assertions in the Code 11](#_Toc366477249)

[6.4.3 Segmentation Faults 11](#_Toc366477250)

[6.4.4 Test Application ‘standalone’ 12](#_Toc366477251)

[6.4.5 Test of Internal States 12](#_Toc366477252)

[6.4.6 Test Sequence Hierarchies 12](#_Toc366477253)

[6.4.7 TEST\_SUITE / TEST\_STEP 12](#_Toc366477254)

[7 Math Tutorial 12](#_Toc366477255)

[7.1 File Overview 12](#_Toc366477256)

[7.2 Starting Point 13](#_Toc366477257)

[7.2.1 Files 13](#_Toc366477258)

[7.2.2 Testrun 13](#_Toc366477259)

[7.3 First Tests 14](#_Toc366477260)

[7.4 Automatic Tests in Eclipse 16](#_Toc366477261)

[7.5 Test of Assertions 17](#_Toc366477262)

[7.6 Grouping 18](#_Toc366477263)

[8 Advanced Tutorial 19](#_Toc366477264)

[8.1 Tutorial Example Code 19](#_Toc366477265)

[8.2 File Overview 19](#_Toc366477266)

[8.3 Starting Point 20](#_Toc366477267)

[8.4 Application Initialization, Our First Mocked Functions 20](#_Toc366477268)

[8.5 Receiving a Zero Length Message, Common Setup Function 21](#_Toc366477269)

[8.6 Using the Callback Registration in uhfInit 23](#_Toc366477270)

[8.7 Receiving a Few Messages that are not Recognized, more Mock Functions 24](#_Toc366477271)

[8.8 Segmentation Faults 26](#_Toc366477272)

[8.9 Assertions or Proper Error Handling? 27](#_Toc366477273)

[8.10 Receiving a Get Version Message 28](#_Toc366477274)

[8.11 Code Coverage 30](#_Toc366477275)

[8.12 Data Driven Testing 31](#_Toc366477276)

[8.13 Checking the Code with Splint 32](#_Toc366477277)

[8.14 Mock or Stub? 34](#_Toc366477278)

[8.15 Debugging a Test Using a Debugger 37](#_Toc366477279)

[9 Duplicate Test Frameworks 39](#_Toc366477280)

[9.1 Initial State 39](#_Toc366477281)

[9.2 Connecting the new Framework 39](#_Toc366477282)

# Revision History

|  |  |  |  |
| --- | --- | --- | --- |
| Revision | Date | Changed by | Description |
| PA1 | 2005-09-19 | Magnus Ivarsson | Document draft |
| PA2 | 2010-08-02 | Magnus Ivarsson | Update for new mock utilities |
| PA3 |  | Magnus Ivarsson |  |
| PA4 | 2010-08-11 | Magnus Ivarsson | Added uthTestGetActualParam example |
| PA5 | 2011-05-09 | Magnus Ivarsson | Adapted the old tutorial to the updated icTest system |
| PA6 | 2011-05-17 | Magnus Ivarsson | Updated after review and added a section on splint |
| PA7 | 2011-05-18 | Magnus Ivarsson | Added build automatically in eclipse and more splint |
| PA8 | 2011-05-20 | Magnus Ivarsson | Added section on stubs |
| PA9 | 2012-09-20 | Kyle Moreau | Updated to new ISIS template. |
| PA10 | 2013-09-01 | Magnus Ivarsson | Updated svn references |

# Introduction

This document is an introduction to the icTest unit test system used in the MBS build environment. The icTest system is an update of the system used in Hi-O and Aperio. It is intended for firmware developed in the C language.

## Purpose

This document is written to help developers understand the test setup, first by a brief description of the setup and then by some tutorials.

Feel free to make suggestions, bug reports and ask questions:

magnus@unitware.se, +46 730 886100

## Scope

## Definitions and abbreviations

|  |  |
| --- | --- |
| Expression | Description |
| Black box unit test | A test that is designed from the interface of the unit to test. The internals of the unit is considered a black box. Ideally a change of unit implementation with unmodified interface should require no update in the unit test. |
| DUT | “device under test” – the unit that is tested, in this case often a c-file or a function within a c-file. |
| Fake | Here: A faked C file that replaces a real C file that the unit under test is dependent on, it is an implementation of a function interface needed to compile the test.  A faked function can be configured as a stub or a mock. If a fake function is called when not expected the test will fail. |
| Mock | A fake function that will fail the test if the parameters differ from the expected values, or if it is not called. |
| Stub | A fake function that is expected to be called, but is not relevant to the actual test. A stub will not fail a test. |
| Test double | Synonymous for fake |
| TBD | To Be Defined |
| Unit | Here: The lowest level of source code packaging for test, change and version management. Typically made up of one C header file and one C code file. |
| White box unit test | A test that is designed with knowledge of the internals of the unit to test. Change of unit implementation will require update of the unit test. |

## References

|  |  |
| --- | --- |
|  | Test Driven Development for Embedded-C, James W. Grenning, 978-1-934356-62-3. A good book on the subject with examples written in C. Excellent section on refactoring in section III. |
|  | Splint Manual, <http://www.splint.org/manual/>. A manual for the static code analysis tool splint. |

# Motivation for Unit Tests and Test Driven Development

There are many reasons to work with test driven development, using the unit tests as a method to make sure that the correct code is written and that it stay correct. Have a look in the references for a few good books on the subject.

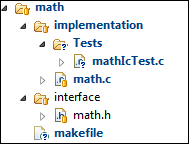
The main reasons for using unit tests are:

* Unit test of software modules let the developer see and fix bugs quite early in the development process; this saves a lot of time and frustration and minimizes the debugging time.
* Unit test of software modules makes it easier to catch errors introduced when maintaining old code or adding new functionality.
* Using unit tests will guarantee that the code is testrun early and often, in an environment that is easy and fast to run and debug. The small units makes it easy to pinpoint the error since it is so little code running in each case.
* During the development of a module and the modules lifecycle the amount of tests will grow, they remain in the test sequence and will notify anyone that is working in the module that some change had unwanted side effects. The tests that you write can be regarded as insurance that nobody will wreck the functionality that you once have created without noticing.

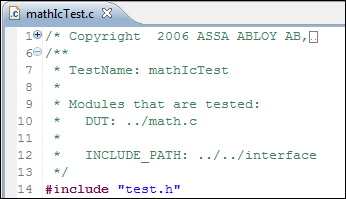
Unit tests that are written prior to implementation of a module help the developer to focus on interfaces and modularity. Addressing these issues in an early stage is important since changes are cheaper and easier to make then. This also helps the developer to focus on the essential functionality without adding to much functionality that might be needed later on. If that is needed then it will be added eventually – including appropriate tests.

# Unit Test Example

In the icTest framework a unit test is a C-file located in a directory called Tests and that have a name ending with IcTest.c.



The icTest file also has a few mandatory elements located in a comment which is used by the framework to set up the test build.



In this case the DUT is the file math.c located in the parent directory relative to the test file. The include path is also specified relative to the test file in this case. The paths are just copied to the generated makefile so any valid makefile syntax including variables can be used.

The workflow is simply:

1. Get an idea of the functionality required by the module and write the API in source.h
2. Run the empty test to see that the test system is up and running.
3. Write a test function in testSource.c that uses the source.h API to test the wanted functionality.
4. Run the test and make sure it fails (a pass at this time means that there might be an error in the test).
5. Implement the production code until the tests passes.
6. Refactor the code and the test (but not at the same time) and verify that the test still pass
7. Repeat steps 4-7

The idea is to test without the need to add test-specific code into the DUT. (Black box testing), this allows for changes in the implementation without changes in the test code.

The test functions are executing the tests, calling the functions in source.c to verify that it is responding as expected. The expected results are verified with the testAssert macros, if functions are expected to be called in other modules then these are added as mocks and the expected call sequence is specified in the test function prior to the actual call.

Any test assertion that fails, any call to a non expected mock function or if an expected mock function is left uncalled will fail the test.

# A Walkthrough of the Files

## Tests/nnnIcTest.c

Contains the test specification located in a comment. Certain keywords is found by the test system and a test application is generated accordingly.

/\*\*

\* TestName: radioHandlerIcTest

\*

\* Modules that are tested:

\* DUT: ../radioHandler.c

\*

\* INCLUDE\_PATH: $(call mbs-component-interface, radio)

\*

\* MOCK: $(call mbs-component-interface, uhf)/uhf.h

\*/

The keywords that can be used (and are reserved words, even in comments) are:

|  |  |
| --- | --- |
| TestName: | The name of the test, must be the same as the filename |
| DUT: | The file(s) that are under test |
| ADDITIONAL\_MODULE: | Any module that shall be compiled into the test but not treated as a DUT. For example manually created test doubles |
| MOCK: | A header file that shall be faked with the mock generator |
| INCLUDE\_PATH: | Path(s) to header files that is not located at the DUT, Test, or Mocks |
| DEFINE: | Any defines that shall be set by the compiler (-D will be added) |
| TEST\_FUNCTION( name ) | A test function |
| TEST\_SETUP() | A common setup function, run before all test functions |

A common test setup function can be used, it shall be named TEST\_SETUP().

TEST\_SETUP()

{

...

}

Test functions are defined with help of the macro TEST\_FUNCTION( testName )

TEST\_FUNCTION( testMe )

{

int a = 1;

testAssertEqual( 1, a );

}

TEST\_FUNCTION( testAnotherThing )

{

//yes this will probably fail

testAssert( !strcmp( "gorilla", "a kind of monkey" ) );

}

The macros testAssertEqual() and testAssert() are used test an assertion. If it fails the user is informed of the error cause and the place of the test that failed (filename and line). The test function is aborted and the test runner continues with the next test function.

## Tests/module.mk

*Obsolete, these files are now generated automatically.*

## TestDouble / Fake

A test double (or fake) is an “empty” module that implements an API that is needed by the module in test. It provides the possibility to verify that the correct functions are called by the function under test, to verify the arguments and to supply return values that which shall be given back to the tested code.

A function that is faked will fail the test if it is called without being enabled in the test. A fake can be enabled as a Stub or a Mock, the difference is that stubs will never fail the test, but needs to be active since they are called by the DUT for whatever reason. A mock on the other hand must be called in the correct order and with correct parameters – or the test will fail.

The icTestSuite will generate the necessary fake-files if you specify the MOCK: keyword in the test comment. If the header file is hard to parse then it is also possible to create a fake manually and add it as an additional module to be included in the test with the ADDITIONAL\_MODULE: keyword.

A test specification comment might have the following row:

/\* MOCK: $(TOP)/uhf/uhf.h \*/

A fake file will then be generated and a function might look like this:

U8 \* uhfAlloc( U8 bufSize )

{

icdFakeEnter( uhfAlloc );

icdFakeParam( 0, bufSize );

return icdFakeExitReturnValue( U8 \* );

}

The function names used in the fake all starts with icdFake. Do not forget to call the icdFakeExitReturnValue( <type> ) or icdFakeExit() if no return value shall be returned, it is in theese functions that the evaluation of the call takes place.

The test might contain these rows, simulating that no memory were available, note that the expectations are set up before the DUT is called:

icdTestExpectMock( uhfAlloc );

icdTestParam( 0, SIZEOF\_RESPONSE\_MESSAGE );

icdTestReturnValue( NULL );

..

callToTheDut(..)

The functionnames (actually macros) used in the test all start with icdTest, Look in the file icDouble.h for documentation of these functions.

## Running the Test

The make system is used to compile, link and run the test application, output is written to standard output and a coverage report is generated. If run in teamcity some additional reports are generated to update the test count statistics.

To start the test browse to the directory containing the makefile and type make icTest.

make **icTest**

If you want to run a specific test only then icTest can be replaced with the test name:

make **mathIcTest**

When there is a lot of tests the verbose output to stdout can be tiring, to improve the test speed the tests can be run with the \_silent suffix

make icT**est\_silent**

make **mathIcTest\_silent**

And for Hi-O it is also possible to generate a coverage report

make **icTest\_coverage**

make **mathIcTest\_coverage**

Details

### Test Assertions

A number of test assertions are available, please consult the test.h for an updated instruction.

The most used are testAssert and testAssertEqual. The error output contains a copy of the assertion, if you want to provide a better explanation on what went wrong the testAssertStr( assertion, ( prf )) can be used where “prf” is a format string and possibly a few arguments like you would use in a printf call. (See the *italic* text below)

a = 2;

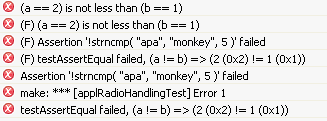
b = 1;

testAssert( !strncmp( "apa", "monkey", 5 ));

testAssertStr( a < b, *("(a == %d) is not less than (b == %d)", a, b)* );

testAssertEqual( a, b );

These rows will give error messages in stdout and is parsed into the eclipse problem view, doubleclicking on them brings you to the correct file.



Some more assertions from the test framework:

|  |  |
| --- | --- |
| testAssert( assertion ) | Fail if not (assertion) |
| testAssertStr( assertion, prf ) | Fail and print the (prf) if not (assertion), a prf is what you would write after a printf.  testAssertStr( 1!=0, ( “apa %d”, 32 )); |
| testAssertEqual( expected, actual ) |  |
| testAssertNotEqual( expected, actual ) |  |
| testAssertNotNull( actual ) |  |
| testAssertMemCmpEqual( expected, was, numOfBytes ) |  |
| testAssertEqualString( expected, was) |  |
| testFail( prf ) |  |
| testTrack(); | Update the last known position, used to find segmentation faults |

### Handling (Testing) Error Assertions in the Code

It is possible to test that the production code has an IC\_ERROR\_ASSERT or MBS\_ASSERT to handle certain errors. This is done with the help of a special macro that wraps the call to the production code. You can think of it as a C-version of try-catch.

TEST\_FUNCTION( **one\_div\_zero\_\_is\_handled\_by\_assertion** )

{

TEST\_EXPECT\_ASSERTION( mathDivide( 1, 0 ) );

}

#include “icAssert.h”

U8 **mathDivide**( U8 dividend, U8 divisor )

{

icAssert( 0 != divisor );

return dividend / divisor;

}

Some icAssertions:

|  |  |
| --- | --- |
| icAssert( assertion ) | Fail if not (assertion) |
| icAssertNotNull( ptr ) | Fail if ptr is NULL |
| icAssertEqual( expected, actual ) | Fail if expected != actual |
| icAssertNotEqual( expected, actual ) | Fail if expected == actual |

### Segmentation Faults

Segmentation faults arise when addressing of illegal memory occurs. Since this can happen anywhere in the program pinpointing the exact location can be tricky. Some help is given telling you where the last testAssert macro where used. To further pinpoint the location without the assertion and string you might want to use the testTrack() macro.

1 testAssert( bogus );  
2 ;  
3 testTrack();  
4 ;  
5 ;  
6 \*(Char\*)0 = 1;  
7 ;

In this case the reported position is line 3 if testTrack() is used or line 1 if not.

### Test Application ‘standalone’

The test application can be called with the following command line options.

-h Give some help information

-o nn output trace to log file nn

-v Turn on more trace output

-vv And then some

### Test of Internal States

Do not try to test the internal stuff of the module, the unit tests are meant to test module behaviour and not implementation. Any attempt to expose internal variables or to create getters affects the code size and violates namespaces and ‘API strictness’. It also makes it hard to refactor the implementation of a module.

### Test Sequence Hierarchies

*Obsolete*

### TEST\_SUITE / TEST\_STEP

*Obsolete*

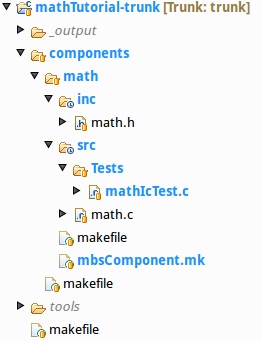
# Math Tutorial

The tutorial illustrates the development of a simple math module using test driven development.

The scope of the math module is limited to illustrate some simple unit test features.

## File Overview

Files involved and the file locations.



|  |  |
| --- | --- |
| makefile  components/makefile components/math/makefile | The makefile(s) includes the build system which in turn finds the relevant tests and how to run them. (The tests located below the makefile directory is run) |
| components/math/src  math.c | File containing the module we want to test |
| components/math/src/Test  mathIcTest.c | File containing the test functions and the test specification markup used to build the actual test application. |
| components/math/inc  math.h | File providing the external interface of the module. |

## Starting Point

### Files

Some files are necessary to have in place in order to start developing.

(Get the project from https://svn.assaabloy.net:8080/svn/mbs/projects/education/unitTestTutorial/mathTutorial/branches/0\_starting\_point)

#### Makefiles

The makefiles must be set up properly; it is done a little bit differently depending on the project structure. It is outside of the scope of this document.

Worth mentioning is that the list of all files that are used in the production source is also used to generate the code coverage report so that files not tested at all shows up as 0%.

#### component/math/implementation/Tests/module.mk

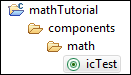
*This file is now obsolete, the corresponding file is autogenerated nowadays.*

#### component/math/implementation/math.c and component/math/interface/math.h

These files are empty at the moment

### Testrun

To test that the project is ready for the tutorial and that the necessary tools are installed use the make target “icTest” in one of the directories contacting a makefile.



The console output should look like this:

> cd components/math  
> make icTest  
make icTest

ls: cannot access /cygdrive/c/work/gitTutorial/mathTutorial/components/math/implementation/Tests/\*IcTest.c: No such file or directory

--all test done-->

The files were compiled and then the test was run, since no tests were added then no tests failed. We are ready to start working!

## First Tests

It is time to start working on our actual code. And since we are cool and like to produce good stuff we go for the test driven development approach: <http://en.wikipedia.org/wiki/Test-driven_development> basically this means that we work like this:

1. write a test that fails
2. write exactly the code that is needed for the test to pass
3. Refactor the code and the test – but never at the same time
4. repeat the above for all new functions that are added or modified

So let’s get started. The first question that arises is of course the most important. What are we supposed to develop? To write a test forces us to think about this very early if you do not know what to test then it’s no good idea to develop it in the first place is there?

This time we shall create a very simple math function that adds two numbers and returns the result. Lets write a first test, naming it to reflect what we are trying to do:

#include "math.h"

TEST\_FUNCTION( one\_plus\_one\_is\_two )

{

testAssertEqual( 2, mathAdd( 1, 1 ) );

}

Then we make the project compile by adding the prototype in math.h

/\* Include files ------------------------------------------------------------ \*/

#include "portab.h"

/\*\* add two numbers and return the result

\* \param a

\* \param b

\* \return a+b

\*/

U8 mathAdd( U8 a, U8 b );

and an empty function stub in math.c.

/\* Include files ------------------------------------------------------------ \*/

#include "math.h"

int mathAdd( int a, int b )

{

return 0;

}

Almost there, the test framework needs some information on what to do, like what file that shall be tested and where to find the header file.

/\*\*

\* TestName: mathIcTest

\*

\* Modules that are tested:

\* DUT: ../math.c

\*

\* Include paths:

\* INCLUDE\_PATH: ../../interface

\*/

Running the test gives a failure:

--- compiling object file math/Tests/mathTest.o ---

--- compiling object file math/math.o ---

=== linking ../../../../Bin/test\_mathTest/Windows/mathTest\_appl.exe ====

>> FAILED at math/Tests/mathTest.c:24: testAssertEqual failed, (2 != mathAdd( 1, 1 )) => (2 (0x2) != 0 (0x0))

==================================================================

Regression Test Generator

Version 0.3, built Aug 3 2010, 13:14:39

------------------------------------------------------------------

#<00> 'one\_plus\_one\_is\_two'

>> FAILED at math/Tests/mathTest.c:24: testAssertEqual failed, (2 != mathAdd( 1, 1 )) => (2 (0x2) != 0 (0x0))

*math/Tests/mathTest.c:24: (F) testAssertEqual failed, (2 != mathAdd( 1, 1 )) => (2 (0x2) != 0 (0x0))*

math/Tests/mathTest.c:24: (F) testAssertEqual failed, (2 != mathAdd( 1, 1 )) => (2 (0x2) != 0 (0x0))

### TEST FAILED ########### c:\work\AperioFirmware\_trunk\ic\Bin\test\_mathTest\Windows\mathTest\_appl.exe 1 tests run in 0 nodes, 1 of them failed

make: \*\*\* [mathTest] Error 1

And by adding an implementation in the function the test will pass:

int mathAdd( int a, int b )

{

return 2;

}

Clearly this is not exactly what we would be satisfied with so we add a few more tests, and for every test the implementation is improved.

TEST\_FUNCTION( one\_plus\_one\_is\_two )

{

testAssertEqual( 2, mathAdd( 1, 1 ) );

}

TEST\_FUNCTION( zero\_plus\_one\_is\_one )

{

testAssertEqual( 1, mathAdd( 0, 1 ) );

}

TEST\_FUNCTION( one\_plus\_zero\_is\_one )

{

testAssertEqual( 1, mathAdd( 1, 0 ) );

}

U8 mathAdd( U8 a, U8 b )

{

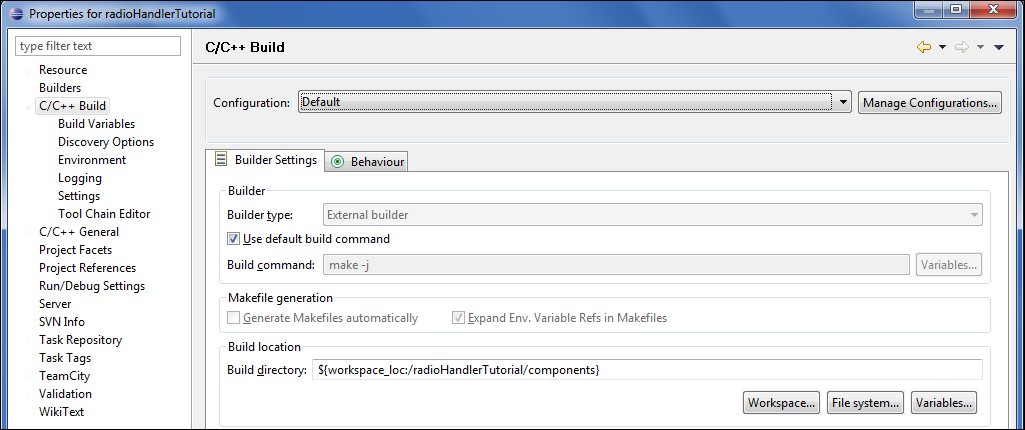
return a + b;

}

(The current code state: [https://svn.assaabloy.net:8080/svn/mbs/projects/education/unitTestTutorial/mathTutorial/branches/1\_add\_implemented](https://svn.hq.assaabloy.org/mbs/projects/education/unitTestTutorial/mathTutorial/branches/1_add_implemented))

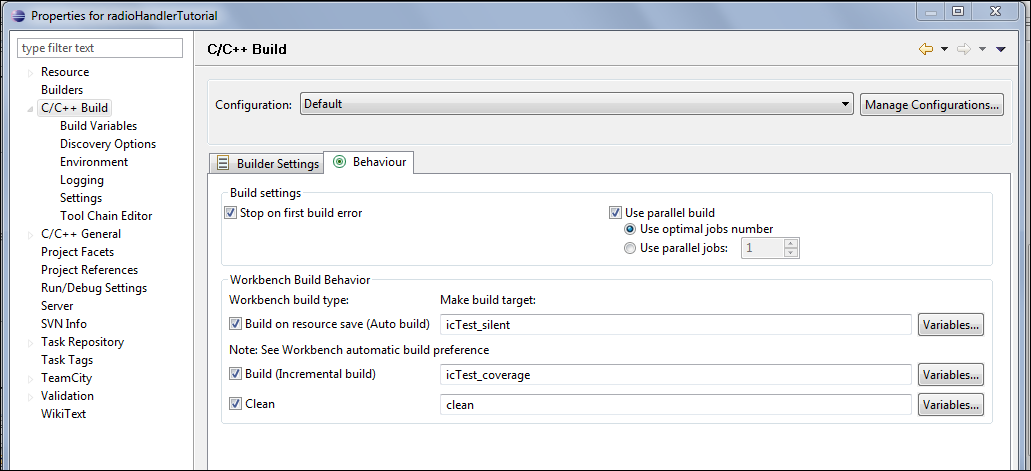
## Automatic Tests in Eclipse

This might be a good time to setup the tests to run at once when saving a file in eclipse. These settings are in the project properties:

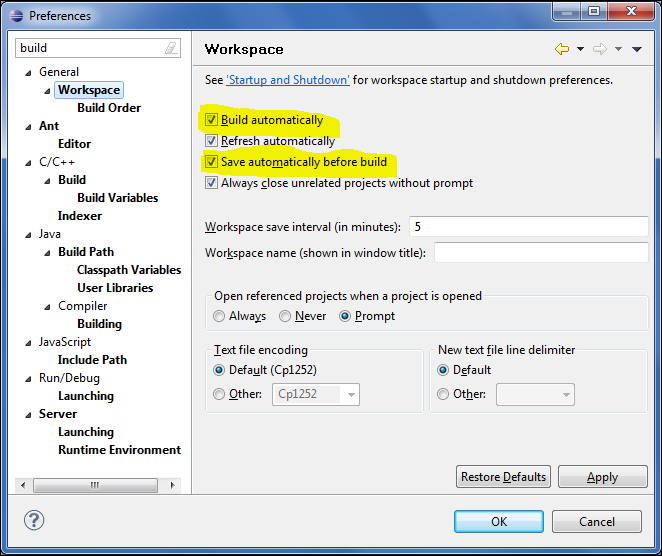


Set the “Build location” to the level you want, the lower you set it in the folder structure the fewer tests ar run. Faster but you might miss some errors that are affecting other parts of the project. For the training we set it to components since all code we write are there. (at the top level there will be some more tests of the unit test framework itself).

Use these settings in the behaviour tab:



And finally check that the workbench is set up for automatic builds (in the Windows->preferences menu):



The project will now be tested at every save, if you press <ctrl-B> the tests are run and the coverage analysis is done.

## Test of Assertions

So far so good, now we shall implement a function that divides two integers, start with a test at a time, I’ll show some at once here for layout reasons.

TEST\_FUNCTION( zero\_div\_\_one\_\_is\_zero )

{

testAssertEqual( 0, mathDivide( 0, 1 ) );

}

TEST\_FUNCTION( one\_div\_\_one\_\_is\_one )

{

testAssertEqual( 1, mathDivide( 1, 1 ) );

}

int mathDivide( int dividend, int divisor )

{

return dividend / divisor;

}

We shall have an assertion to catch the error condition when the divisor is zero. This is to aid in debugging, it is considered the callers responsibility to never call us with zero divisor but just in case we add an assertion that might be removed in production code.

We use the TEST\_EXPECT\_ASSERTION macro to verify that the error condition is properly handled

TEST\_FUNCTION( one\_div\_zero\_\_is\_handled\_by\_assertion )

{

TEST\_EXPECT\_ASSERTION( mathDivide( 1, 0 ) );

}

Test that the test fails and then add the following row in the implementation

#include “icAssert.h”

U8 mathDivide( U8 dividend, U8 divisor )

{

icAssert( 0 != divisor );

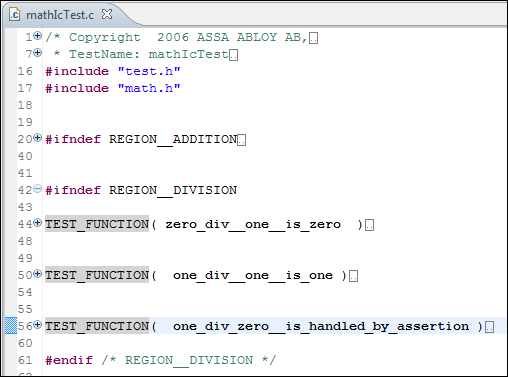
return dividend / divisor;

}

## Grouping

When there are lots of test functions or if they belong to different aspects of the DUT it’s a good practice to group them in either different test files, named properly to reflect the aspect. (mathAdditionIcTest.c, mathDivisionIcTest.c).

Another way is to group the contents in the same file using #ifndef:s with the REGION\_\_ prefix. Then the folding function of eclipse (and other editors) can be used to collapse the parts that we are not working on at the moment.



Which one to use would depend on the overall size, and if different setup code needs to be used.

(the code at this final stage can be found here: <https://svn.assaabloy.net:8080/svn/mbs/projects/education/unitTestTutorial/mathTutorial/branches/2_div_implemented>)

# Advanced Tutorial

## Tutorial Example Code

The example illustrates a test of an application handling radio requests. The test setup shall use a fake to emulate the uhf software module, focusing on the application handling of messages.

The function that we will develop is the following:

1. When the radioHandler is initiated the uhf module shall be initiated too.
2. When a message is received it shall be parsed, if the message id is unknown then an error response shall be sent.
3. If the message Id is a parameter get version then a appropriate radio message be sent.
4. Radio messages shall be allocated (uhfAlloc) prior to use
5. Received radio messages shall be released (uhfFree) after use

## File Overview

Files involved and the file locations.



|  |  |
| --- | --- |
| makefile  components/makefile components/radio/makefile | The makefile(s) includes the build system which in turn finds the relevant tests and how to run them. (The tests located below the makefile directory is run) |
| components/radio/src  radioHandler.c | File containing the module we want to test |
| radioMessages.h  version.h | Files containing the radio protocol format and some version constant numbers. |
| components/radio/src/Test  radioHandlerIcTest.c | File containing the test functions and the test specification markup used to build the actual test application. |
| components/radio/inc  radioHandler.h | File providing the external interface of the module. |
| components/uhf/inc/uhf.h | File providing the interface of the uhf module. |

## Starting Point

The starting point for this tutorial is found here: <https://svn.assaabloy.net:8080/svn/mbs/projects/education/unitTestTutorial/radioHandlerTutorial/branches/0_starting_point>

Please look in the previous tutorial for a brief understanding of the structure and contents of the various files.

Run the tests and verify that all is compiling and that the test application is run without errors even if no tests are added.

## Application Initialization, Our First Mocked Functions

When radioInit() is called the uhfInit shall be called. We will add a test that is calling radioInit() and also create the faked uhfInit(). Then we will see the test fail and finally add the needed implementation in radioHandler.c.

Note that we tell the test to expect a mock. The test will fail if it is not called or called to many times.

The fake/test double functionality is added by including the icDouble.h, and please make an effort to name the test so the intention is clear even after a few months.

/\*\*

\* TestName: radioHandlerIcTest

\*

\* DUT: ../radioHandler.c

\*

\* INCLUDE\_PATH: $(call mbs-component-interface, radio)

\*

\* MOCK: $(call mbs-component-interface, uhf)/uhf.h

\*/

#include "test.h"

#include "icDouble.h"

#include "radioHandler.h"

#include "uhf.h"

TEST\_FUNCTION( when\_calling\_radioInit\_uhfInit\_shall\_be\_called )

{

icdTestExpectMock( uhfInit );

radioInit();

}

A test run will tell us that uhfInit is not implemented. Adding the uhf.h to the MOCK: keyword in the test comment will tell the mock generator to create the fake.

\* MOCK: $(call mbs-component-interface, uhf)/uhf.h

The test runs, but the expected function is not called.

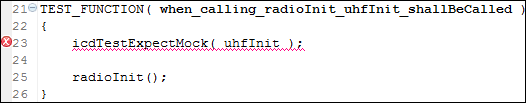
-~- radioHandlerIcTest: running radioHandlerIcTest -~-

>> FAILED at **/cygdrive/c/work/gitTutorial/radioHandlerTutorial/components/radio/implementation/Tests/radioHandlerIcTest.c:23: Not all expected functions called. (1 function was not called)**

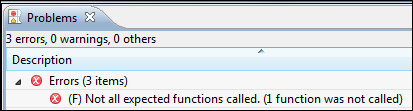
…

### TEST FAILED ###########

The place where the function was expected is also highlighted in the editor



and in the problem view:



Adding the implementation in radioInit() solves the problem, for now we send NULL as the callback parameter.

**void** **radioInit**( **void** )

{

uhfInit( NULL );

}

(the code state is now: [https://svn.assaabloy.net:8080/svn/mbs/projects/education/unitTestTutorial/radioHandlerTutorial/branches/1\_radioInit\_first\_implementation](https://svn.hq.assaabloy.org/mbs/projects/education/unitTestTutorial/radioHandlerTutorial/branches/1_radioInit_first_implementation))

## Receiving a Zero Length Message, Common Setup Function

The uhf module is not allowed to forward zero length or null pointer messages; if it does we want to make sure that the developers are notified rather than working on data that is invalid.

We add two tests see that they fail and implement the assertion checks:

And we take a shortcut with the uhfRxCb callback, we’ll get back to handling registration of callback pointers and such later on. For now we add a function **void** **radioRxCb**( U8 length, U8 \* message ) in radioHandler.h.

TEST\_FUNCTION( **when\_receiving\_a\_zero\_length\_message\_\_an\_assertion\_is\_trigged** )

{

//make sure the test starts from a known state

icdTestExpectMock( uhfInit );

radioInit();

//start the actual test:

// we expect the application to fail on an assertion, the uhf module

// is not allowed to forward zero length messages

{

RadioMessage rxMsg;

TEST\_EXPECT\_ASSERTION( radioRxCb( 0, (U8\*)&rxMsg ) );

}

}

TEST\_FUNCTION( **when\_receiving\_a\_null\_pointer\_message\_\_an\_assertion\_is\_trigged** )

{

//make sure the test starts from a known state

icdTestExpectMock( uhfInit );

radioInit();

//start the actual test:

// we expect the application to fail on an assertion, the uhf module

// is not allowed to forward NULL pointer messages

{

TEST\_EXPECT\_ASSERTION( radioRxCb( 1, NULL ) );

}

}

**#include** "icAssert.h"

**void** **radioRxCb**( U8 length, U8 \* message )

{

icAssert( 0 < length );

icAssert( NULL != message );

}

Now the tests are green and looking at the tests we note that all share the same initialization code. This can be moved to a common setup function that is registered in the test start list.

In this simple test I feel that it’s ok to have validating code in the setup function, but if it gets complicated then it’s better to hav that code in a separate test. Now the test file looks like this:

TEST\_SETUP() //when\_calling\_radioInit\_uhfInit\_shallBeCalled

{

icdTestExpectMock( uhfInit );

radioInit();

}

TEST\_FUNCTION( **when\_receiving\_a\_zero\_length\_message\_\_an\_assertion\_is\_trigged** )

{

RadioMessage rxMsg;

TEST\_EXPECT\_ASSERTION( radioRxCb( 0, (U8\*)&rxMsg ) );

}

TEST\_FUNCTION( **when\_receiving\_a\_null\_pointer\_message\_\_an\_assertion\_is\_trigged** )

{

TEST\_EXPECT\_ASSERTION( radioRxCb( 1, NULL ) );

}

(the code state is now: [https://svn.assaabloy.net:8080/svn/mbs/projects/education/unitTestTutorial/radioHandlerTutorial/branches/2\_common\_setup](https://svn.hq.assaabloy.org/mbs/projects/education/unitTestTutorial/radioHandlerTutorial/branches/2_common_setup))

## Using the Callback Registration in uhfInit

Lets deal with the temporary rx callback that we added in the last section. The uhf module expects a callback to be registered in the call to uhfInit(..). This means that there is a loose coupling between the modules and that the implementation of the callback can (and should) be hidden (that is declared as a local static function).

In this case the test needs to use a parameter value that has been given to a fake by the production code. For this the fake expectation can be extended with a call to icdTestGet() macro to fetch a registered callback function pointer that we’ll use for to test reception of messages. The icDouble API is documented in the icDouble.h file.

/\*\* get the actual value of a parameter that was sent to a fake

\*

\* \example

\* getting a function pointer to a hidden static function that was registered by the code under test

\*

\* \code

\* typedef void ( \* Callback ) ( void );

\* void registerCallback( Callback cb );

\* void init( void );

\*

\* TEST\_FUNCTION(..)

\* {

\* Callback callbackParameterToGet = NULL;

\*

\* icdTestExpectMock( registerCallback );

\* icdTestParamGet( 0, callbackParameterToGet );

\*

\* testAssertEqual( NULL, callbackParameterToGet );

\* init();

\*

\* icdTestVerifyAllExpectationsAreMet();

\*

\* // Now the callbackParameterToGet has been initialized and can be used to

\* // invoke the hidden static callback function inside the code under test

\* testAssertNotEqual( NULL, callbackParameterToGet );

\* callbackParameterToGet();

\* ...

\* }

\* \endcode

\* \hideinitializer

\*/

**#define** icdTestParamGet( paramNumber, parameter ) …

The test:

**static** UhfRxCb uhfRxCb;

TEST\_SETUP() //when\_calling\_radioInit\_uhfInit\_shallBeCalled and a callback registered

{

icdTestExpectMock( uhfInit );

icdTestParamGet( 0, uhfRxCb );

radioInit();

testAssertNotNull( uhfRxCb );

}

The test fails on the null assertion:



Now the application will need a little update, add the “static” before the radioRxCb and register the function in uhfInit instead, we’ll rename the function as well to make it more clear that the test is not calling it directly:

/\* Static function definitions ---------------------------------------------- \*/

**static** **void** **rxCallback**( U8 length, U8 \* message )

{

icAssert( 0 < length );

icAssert( NULL != message );

}

/\* Module interface function definitions ------------------------------------ \*/

**void** **radioInit**( **void** )

{

uhfInit( rxCallback );

}

## Receiving a Few Messages that are not Recognized, more Mock Functions

Now it’s time for the application to receive a radio message that it does not recognize.

TEST\_FUNCTION( **when\_receiving\_an\_unkown\_message\_\_a\_result\_message\_shall\_be\_returned\_with\_result\_code\_\_unknown\_message\_id** )

{

// we expect the application to allocate a response message,

// send it with the proper contents

// and finally release the received message buffer

RadioMessage rxMsg;

RadioMessage txMsg;

**const** RadioMessage expectedTxMsg = { *RADIO\_MSG\_ID\_\_RESULT*,

{ { *RADIO\_MSG\_RES\_\_UNKOWN\_MSG\_ID* } } };

rxMsg.msgId = *RADIO\_MSG\_ID\_\_UNKNOWN*;

memset( &txMsg, 0xaa, **sizeof**( RadioMessage ));

icdTestExpectMock( uhfAlloc );

icdTestParamImmediate( 0, **sizeof**( RadioMsgId ) + **sizeof**( RadioMessageResult ) );

icdTestReturnValue( &txMsg );

icdTestExpectMock( uhfTx );

icdTestParamImmediate( 0, **sizeof**( RadioMsgId ) + **sizeof**( RadioMessageResult ) );

icdTestParamPtrValue( 1, &txMsg );

icdTestParamPtrBlob( 1, &expectedTxMsg, **sizeof**( RadioMsgId )

+ **sizeof**( RadioMessageResult ) );

icdTestExpectMock( uhfFree );

icdTestParamPtrValue( 0, &rxMsg );

uhfRxCb( 1, (U8\*)&rxMsg );

}

Let’s see what the test does.

First we remember that all tests have a common setup function that initializes the application. Then two RadioMessage:s are allocated (on the stack), one that we shall use to “send” to the application and one that the application will get when it is allocating a message to send. There is also an “expectedMessage” that we will use to compare and verify that the sent message was correct.

The rxMsg is initialized to be of a message type that is not known to the radio handler and the contents of the txMsg is set to a pattern that is not zero and not likely to be the expected message. (like it would be as uninitialized from the heap).

Then we get to the part where we specify the expected order of calls to the mock functions. Note that we also specify the expected parameters and that we set the return value to be given back to the application. Pay special attention to the expected message payload for the uhfTx call.

And when this test fails we add the first implementation:

**static** **void** **rxCallback**( U8 length, U8 \* message )

{

icAssert( 0 < length );

icAssert( NULL != message );

{

RadioMessage \* txMsg = (RadioMessage \*)uhfAlloc( **sizeof**( RadioMsgId )

+ **sizeof**( RadioMessageResult ) );

txMsg->msgId = *RADIO\_MSG\_ID\_\_RESULT*;

txMsg->u.res.resultCode = *RADIO\_MSG\_RES\_\_UNKOWN\_MSG\_ID*;

uhfTx( **sizeof**( RadioMsgId ) + **sizeof**( RadioMessageResult ), (U8\*)txMsg );

}

uhfFree( message );

}

When tests are green we take a minute to refactor the test and the production code – preferably one file at a time or at the least we never refactor the test and the production code at the same time.

One obvious thing here is the frequent calculation of the sizeof( RadioMsgId ) + sizeof( RadioMessageResult ) this can be done once and put in a constant that is named appropriately. Also the constants shoud be all caps. When this is done in the test it looks like so:

TEST\_FUNCTION( **when\_receiving\_an\_unkown\_message\_\_a\_result\_message\_shall\_be\_returned\_with\_result\_code\_\_unknown\_message\_id** )

{

// we expect the application to allocate a response message,

// send it with the proper contents

// and finally release the received message buffer

RadioMessage rxMsg;

RadioMessage txMsg;

**const** RadioMessage EXPECTED\_TX\_MSG = { *RADIO\_MSG\_ID\_\_RESULT*,

{ { *RADIO\_MSG\_RES\_\_UNKOWN\_MSG\_ID* } } };

**const** size\_t SIZEOF\_RESULT\_MESSAGE = **sizeof**( RadioMsgId ) + **sizeof**( RadioMessageResult );

rxMsg.msgId = *RADIO\_MSG\_ID\_\_UNKNOWN*;

memset( &txMsg, 0xaa, **sizeof**( RadioMessage ));

icdTestExpectMock( uhfAlloc );

icdTestParamImmediate( 0, SIZEOF\_RESULT\_MESSAGE );

icdTestReturnValue( &txMsg );

icdTestExpectMock( uhfTx );

icdTestParamImmediate( 0, SIZEOF\_RESULT\_MESSAGE );

icdTestParamPtrValue( 1, &txMsg );

icdTestParamPtrBlob( 1, &EXPECTED\_TX\_MSG, SIZEOF\_RESULT\_MESSAGE );

icdTestExpectMock( uhfFree );

icdTestParamPtrValue( 0, &rxMsg );

uhfRxCb( 1, (U8\*)&rxMsg );

}

The production code can be handled the same way, and in this casa we also break out the code to send the response message to make it clearer what is happening when we add more functions in the next step.

**static** **void** **sendRadioResult**( RadioMsgResultCode resultCode )

{

**const** size\_t SIZEOF\_RESULT\_MESSAGE = **sizeof**( RadioMsgId ) + **sizeof**( RadioMessageResult );

RadioMessage \* txMsg = (RadioMessage \*)uhfAlloc( SIZEOF\_RESULT\_MESSAGE );

txMsg->msgId = *RADIO\_MSG\_ID\_\_RESULT*;

txMsg->u.res.resultCode = resultCode;

uhfTx( SIZEOF\_RESULT\_MESSAGE, (U8\*)txMsg );

}

**static** **void** **rxCallback**( U8 length, U8 \* message )

{

icAssert( 0 < length );

icAssert( NULL != message );

sendRadioResult( *RADIO\_MSG\_RES\_\_UNKOWN\_MSG\_ID* );

uhfFree( message );}

(The code state at this point: [https://svn.assaabloy.net:8080/svn/mbs/projects/education/unitTestTutorial/radioHandlerTutorial/branches/4\_ptrBlob\_refactor](https://svn.hq.assaabloy.org/mbs/projects/education/unitTestTutorial/radioHandlerTutorial/branches/4_ptrBlob_refactor))

## Segmentation Faults

A segmentation fault occurs when the code is accessing memory addresses that is forbidden. The most common case is probably a dereference of a null pointer. In this section we’ll have a look at the help we get from the test system to pinpoint the error. An other way to find the error is of course to start a debugger and run the test application (see **Error! Reference source not found.**).

What would happen if the uhfAlloc() failed and returned NULL? – You guessed right, we’ll write a test to find out and to verify that we have fixed the problem.

TEST\_FUNCTION( **when\_receiving\_an\_unkown\_message\_\_and\_a\_result\_message\_could\_not\_be\_allocated\_\_throw\_assertion** )

{

// we expect the application to try to allocate a response message,

// then throw an assertion when this was not possible

RadioMessage rxMsg;

**const** U8 SIZEOF\_RESULT\_MESSAGE = **sizeof**( RadioMsgId ) + **sizeof**( RadioMessageResult );

rxMsg.msgId = *RADIO\_MSG\_ID\_\_UNKNOWN*;

icdTestExpectMock( uhfAlloc );

icdTestParamImmediate( 0, SIZEOF\_RESULT\_MESSAGE );

icdTestReturnValue( NULL );

TEST\_EXPECT\_ASSERTION( uhfRxCb( 1, (U8\*)&rxMsg ) );

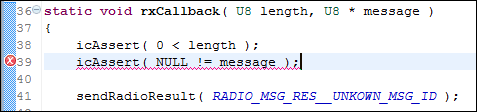
}

Running this will fail the test

>> FAILED at /cygdrive/c/work/gitTutorial/radioHandlerTutorial/components/radio/implementation/radioHandler.c:39: Got a segmentationfault close after this point

### TEST FAILED ###########

And the point is indicated in eclipse like this:



Now this is not the location where the segmentation point occurred, but it is the last point that the test system is aware that has been passed by execution. This “last known point” is updated in most of the test, debug and errorAssertions. While searching a macro testTrack() can be added at various points – but do not forget to remove it and the #include “test.h” from the production code afterwards.

Let’s add the icAssert( NULL != txMsg ); to make the test pass.

**static** **void** **sendRadioResult**( RadioMsgResultCode resultCode )

{

**const** size\_t SIZEOF\_RESULT\_MESSAGE = **sizeof**( RadioMsgId ) + **sizeof**( RadioMessageResult );

RadioMessage \* txMsg = (RadioMessage \*)uhfAlloc( SIZEOF\_RESULT\_MESSAGE );

icAssertNotNull( txMsg );

txMsg->msgId = *RADIO\_MSG\_ID\_\_RESULT*;

txMsg->u.res.resultCode = resultCode;

uhfTx( SIZEOF\_RESULT\_MESSAGE, (U8\*)txMsg );

}

**static** **void** **rxCallback**( U8 length, U8 \* message )

{

icAssert( 0 < length );

icAssertNotNull( message );

sendRadioResult( *RADIO\_MSG\_RES\_\_UNKOWN\_MSG\_ID* );

uhfFree( message );

}

(The code state is now: [https://svn.assaabloy.net:8080/svn/mbs/projects/education/unitTestTutorial/radioHandlerTutorial/branches/5\_segmentation\_faults](https://svn.hq.assaabloy.org/mbs/projects/education/unitTestTutorial/radioHandlerTutorial/branches/5_segmentation_faults))

## Assertions or Proper Error Handling?

But wait a minute! Adding an assertion for the case that uhfAlloc(..) returns NULL is probably not a good idea. Assertions are fine to validate that other developers have done a good job and fulfil the interface specifications - but this is a case we used it in a case that can occur in real life. What would happen if the assertions were not compiled in the production code?

Starting with the test we come up with a better solution, no answer to an unknown message is probably better than an assertion that often trigger a reboot (if the assertion is present, if not then dereferencing a null pointer might lead to real strange side effects).

TEST\_FUNCTION( **when\_receiving\_an\_unkown\_message\_\_and\_a\_result\_message\_could\_not\_be\_allocated\_\_drop\_message\_silently** )

{

RadioMessage rxMsg;

**const** U8 SIZEOF\_RESULT\_MESSAGE = **sizeof**( RadioMsgId )

+ **sizeof**( RadioMessageResult );

rxMsg.msgId = *RADIO\_MSG\_ID\_\_UNKNOWN*;

icdTestExpectMock( uhfAlloc );

icdTestParamImmediate( 0, SIZEOF\_RESULT\_MESSAGE );

icdTestReturnValue( NULL );

icdTestExpectMock( uhfFree );

icdTestParamPtrValue( 0, &rxMsg );

uhfRxCb( 1, (U8\*)&rxMsg );

}

Note that the assertion fails the test with a decent error message and points out the location in the editor.

**static** **void** **sendRadioResult**( RadioMsgResultCode resultCode )

{

**const** size\_t SIZEOF\_RESULT\_MESSAGE = **sizeof**( RadioMsgId )

+ **sizeof**( RadioMessageResult );

RadioMessage \* txMsg = (RadioMessage \*)uhfAlloc( SIZEOF\_RESULT\_MESSAGE );

**if** ( NULL != txMsg )

{

txMsg->msgId = *RADIO\_MSG\_ID\_\_RESULT*;

txMsg->u.res.resultCode = resultCode;

uhfTx( SIZEOF\_RESULT\_MESSAGE, (U8\*)txMsg );

}

}

(The code state is now: [https://svn.assaabloy.net:8080/svn/mbs/projects/education/unitTestTutorial/radioHandlerTutorial/branches/6\_noAssertOnNullAlloc](https://svn.hq.assaabloy.org/mbs/projects/education/unitTestTutorial/radioHandlerTutorial/branches/6_noAssertOnNullAlloc))

## Receiving a Get Version Message

Nothing really new here but we will implement one more function to get the produced code look like something real; When sending a get version message request the application shall respond with a get version response message.

TEST\_FUNCTION( **when\_receiving\_an\_get\_version\_req\_message\_\_a\_get\_version\_rsp\_message\_\_shall\_be\_returned** )

{

// we expect the application to allocate a response message,

// send it with the proper contents

// and finally release the received message buffer

RadioMessage rxMsg;

RadioMessage txMsg;

RadioMessage EXPECTED\_TX\_MESSAGE;

**const** size\_t SIZEOF\_VERSION\_MESSAGE = **sizeof**( RadioMsgId )

+ **sizeof**( RadioMessageVersion );

**const** size\_t SIZEOF\_REQUEST\_MESSAGE = **sizeof**( RadioMsgId );

rxMsg.msgId = *RADIO\_MSG\_ID\_\_GET\_VERSION\_REQ*;

EXPECTED\_TX\_MESSAGE.msgId = *RADIO\_MSG\_ID\_\_GET\_VERSION\_RSP*;

EXPECTED\_TX\_MESSAGE.u.ver.major = VERSION\_MAJOR;

EXPECTED\_TX\_MESSAGE.u.ver.minor = VERSION\_MINOR;

EXPECTED\_TX\_MESSAGE.u.ver.buildNumber = VERSION\_BUILD\_NUMBER;

memset( &txMsg, 0xaa, **sizeof**( RadioMessage ));

icdTestExpectMock( uhfAlloc );

icdTestParamImmediate( 0, SIZEOF\_VERSION\_MESSAGE );

icdTestReturnValue( &txMsg );

icdTestExpectMock( uhfTx );

icdTestParamImmediate( 0, SIZEOF\_VERSION\_MESSAGE );

icdTestParamPtrValue( 1, &txMsg );

icdTestParamPtrBlob( 1, &EXPECTED\_TX\_MESSAGE, SIZEOF\_VERSION\_MESSAGE );

icdTestExpectMock( uhfFree );

icdTestParamPtrValue( 0, &rxMsg );

uhfRxCb( SIZEOF\_REQUEST\_MESSAGE, (U8\*)&rxMsg );

}

And the production code:

**static** **void** **sendGetVersionRsp**( U16 major, U16 minor, U16 buildNumber )

{

**const** size\_t SIZEOF\_TX\_MESSAGE = **sizeof**( RadioMsgId ) + **sizeof**( RadioMessageVersion );

RadioMessage \* txMsg = (RadioMessage \*)uhfAlloc( SIZEOF\_TX\_MESSAGE );

**if** ( NULL != txMsg )

{

txMsg->msgId = *RADIO\_MSG\_ID\_\_GET\_VERSION\_RSP*;

txMsg->u.ver.major = major;

txMsg->u.ver.minor = minor;

txMsg->u.ver.buildNumber = buildNumber;

uhfTx( SIZEOF\_TX\_MESSAGE, (U8\*)txMsg );

}

}

**static** **void** **rxCallback**( U8 length, U8 \* message )

{

RadioMessage \* rxMsg = (RadioMessage \*)message;

icAssert( 0 < length );

icAssertNotNull( rxMsg );

**switch** ( rxMsg->msgId )

{

**case** *RADIO\_MSG\_ID\_\_GET\_VERSION\_REQ*:

sendGetVersionRsp( VERSION\_MAJOR, VERSION\_MINOR, VERSION\_BUILD\_NUMBER );

**break**;

**default**:

sendRadioResult( *RADIO\_MSG\_RES\_\_UNKOWN\_MSG\_ID* );

**break**;

}

uhfFree( (U8\*)rxMsg );

}

To do this we also added a version.h file that is shared between the test and the production code, this to allow for updated values while still testing the logic.

#define VERSION\_MAJOR ( 0x1234 )

#define VERSION\_MINOR ( 0x4567 )

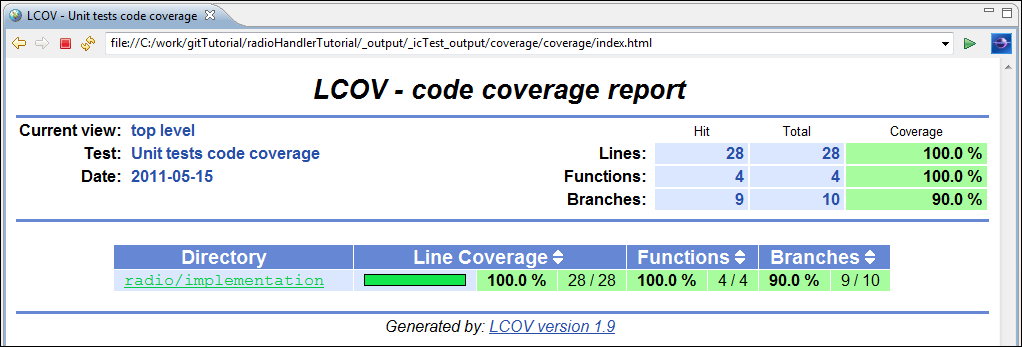
#define VERSION\_BUILD\_NUMBER ( 0x89ab )

(Current state of the code: [https://svn.assaabloy.net:8080/svn/mbs/projects/education/unitTestTutorial/radioHandlerTutorial/branches/7\_noAssertOnNullAlloc\_codeCoverage](https://svn.hq.assaabloy.org/mbs/projects/education/unitTestTutorial/radioHandlerTutorial/branches/7_noAssertOnNullAlloc_codeCoverage))

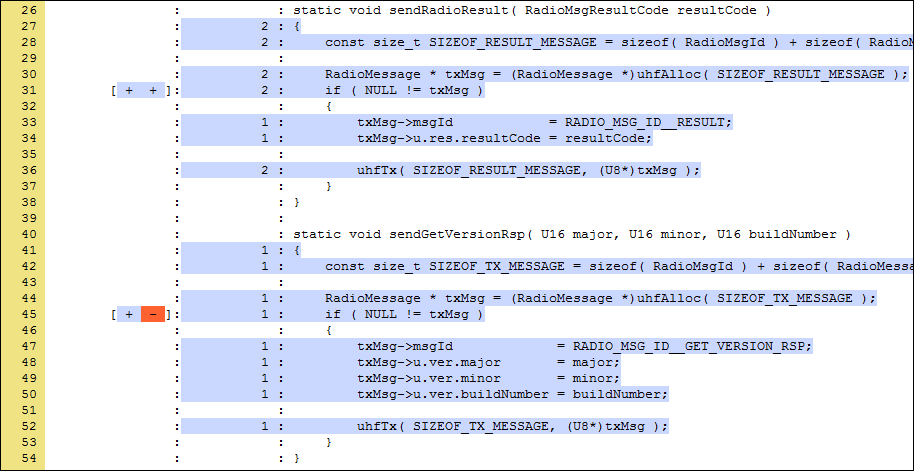
## Code Coverage

The test system can generate a code coverage report. That is which rows and decisions in the DUT that has been executed during the test run. To generate this test run the make target icTest\_coverage.

The result is located in the output directory, for MBS projects: \_output/\_icTest\_output/coverage/coverage.html. open it up in a web-browser



It looks rather good, but there is one decision branch that we have left untested. Can you guess which? Klick on the link radio/implementation and then on radioHandler.c and it will become obvious.



We have not tested the uhfAlloc returned NULL handling of the code. Is this important? Yes, this is an important aspect of the stability of the final code. If the if statement is removed in a coming maintenance then there is nothing that checks this.

The numbers to the left are line numbers, the [+-] is decision branch execution and de next column with numbers tells how many times a certain row has been executed.

## Data Driven Testing

We’ll add another test to check for this null handling.

TEST\_FUNCTION( **when\_receiving\_a\_getVersion\_message\_\_and\_a\_result\_message\_could\_not\_be\_allocated\_\_drop\_message\_silently** )

{

RadioMessage rxMsg;

**const** size\_t SIZEOF\_VERSION\_MESSAGE = **sizeof**( RadioMsgId )

+ **sizeof**( RadioMessageVersion );

**const** size\_t SIZEOF\_REQUEST\_MESSAGE = **sizeof**( RadioMsgId );

rxMsg.msgId = *RADIO\_MSG\_ID\_\_GET\_VERSION\_REQ*;

icdTestExpectMock( uhfAlloc );

icdTestParamImmediate( 0, SIZEOF\_VERSION\_MESSAGE );

icdTestReturnValue( NULL );

icdTestExpectMock( uhfFree );

icdTestParamPtrValue( 0, &rxMsg );

uhfRxCb( SIZEOF\_REQUEST\_MESSAGE, (U8\*)&rxMsg );

}

Whats that smell? Ohh could it be the duplicated code that is starting to grow?

A good reference to codesmells is [**Error! Reference source not found.**], chapter 12 – refactoring.

Yes certainly, the fingers got a little bit itchy the moment I used the copy/paste function just recently. Lets extract a helper function for the two tests handling “uhfAlloc returns null”. Meanwhile we discover that the test was checking the size parameter of the alloc call. This is already tested in the normal usecase and adds unnecessary complexity to this test, be gone!

**static** **void** **helper\_result\_could\_not\_be\_allocated**( RadioMsgId msgId )

{

RadioMessage rxMsg;

**const** size\_t SIZEOF\_REQUEST\_MESSAGE = **sizeof**( RadioMsgId );

rxMsg.msgId = msgId;

icdTestExpectMock( uhfAlloc );

icdTestReturnValue( NULL );

icdTestExpectMock( uhfFree );

icdTestParamPtrValue( 0, &rxMsg );

uhfRxCb( SIZEOF\_REQUEST\_MESSAGE, (U8\*)&rxMsg );

}

TEST\_FUNCTION( **when\_receiving\_an\_unkown\_message\_\_and\_a\_result\_message\_could\_not\_be\_allocated\_\_drop\_message\_silently** )

{

helper\_result\_could\_not\_be\_allocated( *RADIO\_MSG\_ID\_\_UNKNOWN* );

}

TEST\_FUNCTION( **when\_receiving\_a\_getVersion\_message\_\_and\_a\_result\_message\_could\_not\_be\_allocated\_\_drop\_message\_silently** )

{

helper\_result\_could\_not\_be\_allocated( *RADIO\_MSG\_ID\_\_GET\_VERSION\_REQ* );

}

And this brings us to a data driven test. We can loop through all possible values of the MsgId and apply the same test on them all. The downside is that one fail will stop execution and hide the rest, but on the other hand who wants 255, 65535 or more failing tests?

Note the icdTestVerifyAllExpectationsAreMet() call, it makes sure that every iteration is complete with no outstanding expected mocks.

**static** **void** **helper\_result\_could\_not\_be\_allocated**( RadioMsgId msgId )

{

RadioMessage rxMsg;

**const** size\_t SIZEOF\_REQUEST\_MESSAGE = **sizeof**( RadioMsgId );

rxMsg.msgId = msgId;

icdTestExpectMock( uhfAlloc );

icdTestReturnValue( NULL );

icdTestExpectMock( uhfFree );

icdTestParamPtrValue( 0, &rxMsg );

uhfRxCb( SIZEOF\_REQUEST\_MESSAGE, (U8\*)&rxMsg );

icdTestVerifyAllExpectationsAreMet();

}

TEST\_FUNCTION( **when\_receiving\_any\_message\_\_and\_a\_result\_message\_could\_not\_be\_allocated\_\_drop\_message\_silently** )

{

RadioMsgId id;

**for** ( id = 0; id <= *RADIO\_MSG\_ID\_\_MAX*; ++id)

{

helper\_result\_could\_not\_be\_allocated( id );

}

}

Some extraction can be done in the other tests as well, and the test for unknown message id is also suitable for a data driven test that verifies all MsgId:s except those in a list of known MsgId:s. (the latter is left as an exercise for you)

(The current state of the code: [https://svn.assaabloy.net:8080/svn/mbs/projects/education/unitTestTutorial/radioHandlerTutorial/branches/8\_dataDrivenTest](https://svn.hq.assaabloy.org/mbs/projects/education/unitTestTutorial/radioHandlerTutorial/branches/8_dataDrivenTest))

## Checking the Code with Splint

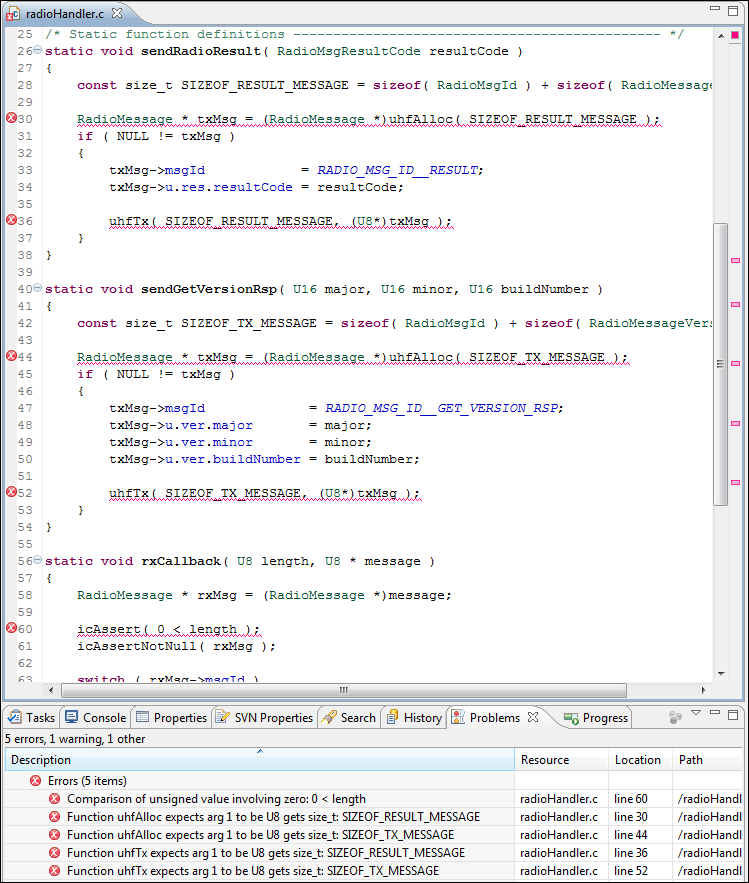
To improve the quality of the code you can enable a static code analysis tool called splint. It’s like the mother of compiler warnings and adds a lot more type checking and will also warn if there is possible uses of null pointers and such.

The manual [**Error! Reference source not found.**] is worth reading, it discusses the kinds of error it looks for and why. The second half is various options to control the checking.

When the environment variable ICTEST\_SPLINT\_ENABLE is set (or MBS\_SPLINT\_ENABLE for MBS projects) then splint is run on the DUT code when it is compiled. In eclipse this can be done by selecting the SPLINT build configuration, or on the command line just type

make MBS\_SPLINT\_ENABLE=yes –j icTest\_silent

On our code it looks like this:



With a few changes there are no more complaints.

…

**const** U8 SIZEOF\_TX\_MESSAGE = (U8)**sizeof**( RadioMsgId ) + (U8)**sizeof**( RadioMessageVersion );

…

icAssert( 0 != length );

…

I agree that this example was a bit lame, but it’s a really useful and powerful tool and it does find real errors, especially memory leaks and stupid data type errors that might inhibit a conditional statement to work. Let’s assume we actually read the manual on the memory model and the annotations we can use to help splint increase the quality.

Lets try out the **/\*@only@\*/** annotation. It tells splint that this parameter is the only pointer to a memory buffer and that the receiver is responsible of freeing it. Another pointer state is **/\*@temp@\*/** which is default for function parameters, (and a few more, rtfm).

Comment out the uhfFree() call in the rxCallback() function and test. The tests hopefully fails, but there is no warning that the memory pointed to by the message pointer has been lost. This is because of the implicit “temp” pointer type of function parameters. If we change the uhf.h a little, adding the :

typedef void ( \* UhfRxCb )( U8 length,   
 ***/\*@only@\*/*** U8 \* message );

void uhfTx( U8 length,   
 **/\*@only@\*/** U8 \* message );

**/\*@only@\*/** U8 \* uhfAlloc( U8 bufSize );

void uhfFree( **/\*@only@\*/** U8 \* buffer );

void uhfInit( UhfRxCb rxCallback );

The result

radio/implementation/radioHandler.c:74:14: Implicitly temp storage message passed as only param (rxMsg aliases message): uhfFree ((U8 \*)rxMsg)

Lets add the **/\*@only@\*/** to the rxCallback pointer parameter aswell.

**static** **void** **rxCallback**( U8 length, /\*@only@\*/ U8 \* message )

Everything is fine, but what if the uhfFree were never called (except that the test would fail)?

radio/implementation/radioHandler.c:75:2: ***Only storage message not released before return***

A memory leak has been detected. Only-qualified storage is not released before the last reference to it is lost. (Use -mustfreeonly to inhibit warning)

radio/implementation/radioHandler.c:56:52: Storage message becomes only

This is a memory leak that might have gone amiss otherwise.

There are also extensive checks against possible use of null pointers and so on..

(The current state of the code: [https://svn.assaabloy.net:8080/svn/mbs/projects/education/unitTestTutorial/radioHandlerTutorial/branches/9\_splint](https://svn.hq.assaabloy.org/mbs/projects/education/unitTestTutorial/radioHandlerTutorial/branches/9_splint))

## Mock or Stub?

There is a test philosophy that there shall be only on assertion per test, in that way every test is focussed on testing one thing an d there is no question what went wrong. If there are several thing s tested in the same test it becomes more fragile, and a failure might shadow the other thing s that should have been tested.

Also tests that have several assertions and mocks tend to be too tightly coupled to the actual implementation of the module – change the order of a few calls and the tests fail even if the relevant behaviour is the same.

Looking at our code we have this situation in some of the tests. When a message is received we test that the responce is allocated sent and that the received buffer is released. Since the mocks must be in order this forces the test to know that the implementation sends the allocates and send the response \_before\_ the rx is freed. If we later on decide that the memory will be better handled if we release the buffer before allocating a new one – then all our tests will fail even if the behaviour is unchanged. (there is also a possibility that we would use the same buffer and return it, we’ll leave that as it is for now).

Ok, if we remove the call to expect uhfFree as a mock then the test fail. A faked function shall fail the test if not mentioned in the test. Because of this there is also a way of enabling a fake as a stub.

A stub is a faked function that allows any number of calls (including no calls) and can be set up to return a specific value. It can even return different return values depending on a set of parameter values which is useful if you for example want to stub a few calls to a parameter get function.

// int paramGet( int parameterNumber );

icdTestEnableStub( paramGet );

icdTestParamImmediate( 0, 0x1010 );

icdTestSetReturnValue( 1 );

icdTestEnableStub( paramGet );

icdTestParamImmediate( 0, 0x1020 );

icdTestSetReturnValue( 2 );

For our test we add tests to test the memory release, stubbing the allocation and transmission.

**static** **void** **helper\_\_rx\_a\_msg\_\_make\_sure\_buffer\_is\_released**( RadioMsgId rxMsgId )

{

RadioMessage rxMsg;

**const** size\_t SIZEOF\_REQUEST\_MESSAGE = **sizeof**( RadioMsgId );

rxMsg.msgId = rxMsgId;

icdTestEnableStub( uhfAlloc );

icdTestEnableStub( uhfTx );

icdTestExpectMock( uhfFree );

icdTestParamPtrValue( 0, &rxMsg );

uhfRxCb( SIZEOF\_REQUEST\_MESSAGE, (U8\*)&rxMsg );

}

TEST\_FUNCTION( **when\_receiving\_an\_unkown\_message\_\_make\_sure\_rxMsg\_buffer\_is\_released** )

{

helper\_\_rx\_a\_msg\_\_make\_sure\_buffer\_is\_released( *RADIO\_MSG\_ID\_\_UNKNOWN* );

}

TEST\_FUNCTION( **when\_receiving\_a\_get\_request\_message\_\_make\_sure\_rxMsg\_buffer\_is\_released** )

{

helper\_\_rx\_a\_msg\_\_make\_sure\_buffer\_is\_released( *RADIO\_MSG\_ID\_\_GET\_VERSION\_REQ* );

}

And then this works we change the original tests to stub the uhfFree call.

TEST\_FUNCTION( **when\_receiving\_an\_unkown\_message\_\_a\_result\_message\_shall\_be\_returned\_with\_result\_code\_\_unknown\_message\_id** )

{

RadioMessage rxMsg;

RadioMessage txMsg;

**const** RadioMessage EXPECTED\_TX\_MESSAGE = { *RADIO\_MSG\_ID\_\_RESULT*,

{ { *RADIO\_MSG\_RES\_\_UNKOWN\_MSG\_ID* } } };

**const** size\_t SIZEOF\_RESULT\_MESSAGE = **sizeof**( RadioMsgId )

+ **sizeof**( RadioMessageResult );

**const** size\_t SIZEOF\_REQUEST\_MESSAGE = **sizeof**( RadioMsgId );

rxMsg.msgId = *RADIO\_MSG\_ID\_\_UNKNOWN*;

icdTestEnableStub( uhfFree );

expectAllocAndTx( SIZEOF\_RESULT\_MESSAGE, &txMsg, &EXPECTED\_TX\_MESSAGE );

uhfRxCb( SIZEOF\_REQUEST\_MESSAGE, (U8\*)&rxMsg );

}

TEST\_FUNCTION( **when\_receiving\_an\_get\_version\_req\_message\_\_a\_get\_version\_rsp\_message\_\_shall\_be\_returned** )

{

RadioMessage rxMsg;

RadioMessage txMsg;

RadioMessage EXPECTED\_TX\_MESSAGE;

**const** size\_t SIZEOF\_VERSION\_MESSAGE = **sizeof**( RadioMsgId ) + **sizeof**( RadioMessageVersion );

**const** size\_t SIZEOF\_REQUEST\_MESSAGE = **sizeof**( RadioMsgId );

rxMsg.msgId = *RADIO\_MSG\_ID\_\_GET\_VERSION\_REQ*;

EXPECTED\_TX\_MESSAGE.msgId = *RADIO\_MSG\_ID\_\_GET\_VERSION\_RSP*;

EXPECTED\_TX\_MESSAGE.u.ver.major = VERSION\_MAJOR;

EXPECTED\_TX\_MESSAGE.u.ver.minor = VERSION\_MINOR;

EXPECTED\_TX\_MESSAGE.u.ver.buildNumber = VERSION\_BUILD\_NUMBER;

icdTestEnableStub( uhfFree );

expectAllocAndTx( SIZEOF\_VERSION\_MESSAGE, &txMsg, &EXPECTED\_TX\_MESSAGE );

uhfRxCb( SIZEOF\_REQUEST\_MESSAGE, (U8\*)&rxMsg );

}

Note that the stubs must be added prior to the mocks, this is to help structuring the tests in a standard way.

We also change the tests that were handling the out of memory condition to set uhfAlloc as a stub returning NULL.

**static** **void** **helper\_result\_could\_not\_be\_allocated\_\_just\_free\_rxBuf**( RadioMsgId msgId )

{

RadioMessage rxMsg;

**const** size\_t SIZEOF\_REQUEST\_MESSAGE = **sizeof**( RadioMsgId );

rxMsg.msgId = msgId;

icdTestExpectMock( uhfFree );

icdTestReturnValue( &rxMsg );

uhfRxCb( SIZEOF\_REQUEST\_MESSAGE, (U8\*)&rxMsg );

icdTestVerifyAllExpectationsAreMet();

}

TEST\_FUNCTION( **when\_receiving\_any\_message\_\_and\_a\_result\_message\_could\_not\_be\_allocated\_\_drop\_message\_silently** )

{

RadioMsgId id;

icdTestEnableStub( uhfAlloc );

icdTestReturnValue( NULL );

**for** ( id = 0; id <= *RADIO\_MSG\_ID\_\_MAX*; ++id)

{

helper\_result\_could\_not\_be\_allocated\_\_just\_free\_rxBuf( id );

}

}

Now we can see what happens if we remove the uhfFree call, and if it is possible to move the uhfFree call to the beginning of the rxCallback.

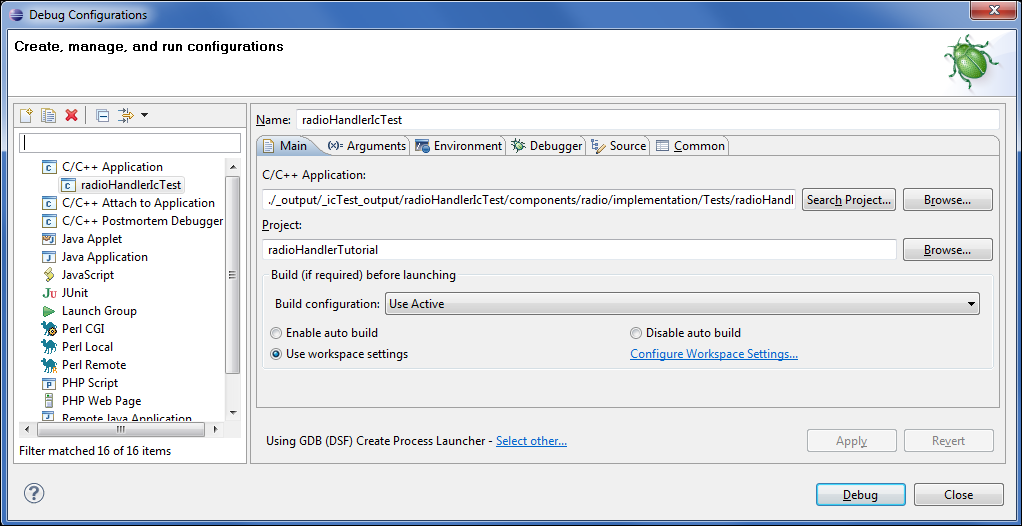
(The current state of the code: [https://svn.assaabloy.net:8080/svn/mbs/projects/education/unitTestTutorial/radioHandlerTutorial/branches/10\_stubs](https://svn.hq.assaabloy.org/mbs/projects/education/unitTestTutorial/radioHandlerTutorial/branches/10_stubs))

## Debugging a Test Using a Debugger

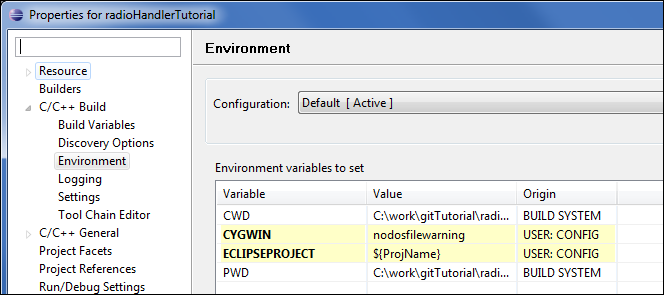
"Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it." --Brian Kernighan

I have found that writing unit tests really limits the time I need to spend debugging code. Not that I do not fire up the debugger – but that is more often just to see things running in real hardware, not so often to find out what I made wrong in my last ten lines of code.

But there are occasions and debugging the test code on the host is a good first step before we try it out in the real hardware. To aid in this there are debug configurations generated on the fly for all test applications.

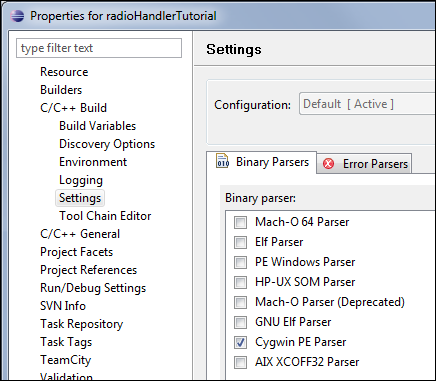


To enable this in Eclipse the project properties must define the ECLIPSEPROJECT environment variable and it shall contain the ${projName} eclipse variable.



If this does not work it is possible to start gdb in a command shell and run the test application from there. Use the set args=-vv to get the verbose output.

Also check that the binary discovery setting is checked in the project properties if you experience problems starting the debug session:



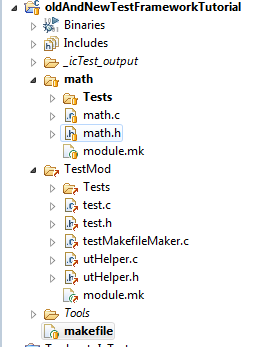
# Duplicate Test Frameworks

In the transition to the icTestSuite it is preferable to have the old and the new test framework active at the same time. This is possible with a few modifications to the makefile.

This tutorial will guide you on the steps of integrating the icTestSuite in a legacy application source tree.

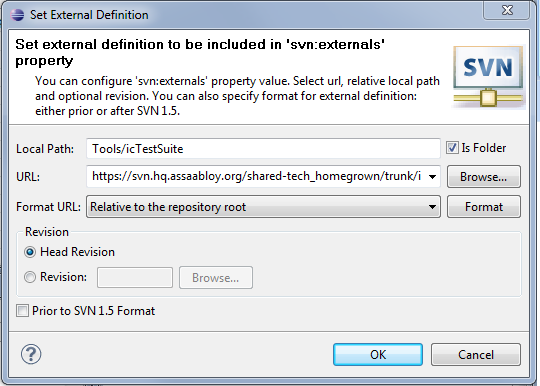
## Initial State

A simple stub project with one module and one test file is used for the tutorial. The test is driven in the old fashioned way using a TestMod module in parallel to the source modules and with hand coded test makefiles.



## Connecting the new Framework

Right click on the project and select team -> set external



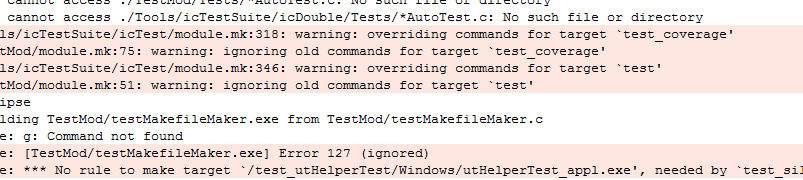
Local Path: Tools/icTestSuite

URL: [https://svn.assaabloy.net:8080/svn/shared-tech\_homegrown/trunk/icTestSuite/development/Tools/icTestSuite](https://svn.hq.assaabloy.org/shared-tech_homegrown/trunk/icTestSuite/development/Tools/icTestSuite)

Lets add the test module in the makefile



Some errors have been introduced:



Now edit the makefile again; the new framework have some variables that can be used to rename the conflicting targets. These are documented in the icTestSuite/icTest/module.mk file

Please contact magnus@unitware.se for any questions on how to do this.