Writing Tests for Managed Code

Contents

[Test Overview 1](#_Toc410733152)

[Test Components 1](#_Toc410733153)

[Test Code 2](#_Toc410733154)

[Test Description File 2](#_Toc410733155)

[Test Parameters 4](#_Toc410733156)

[Test Script 6](#_Toc410733157)

[Test Analysis Script 8](#_Toc410733158)

[Test results 10](#_Toc410733159)

[Test hookup file 11](#_Toc410733160)

[Test event definition file 11](#_Toc410733161)

[Writing great tests 12](#_Toc410733162)

[Further Reading 12](#_Toc410733163)

[Glossary 12](#_Toc410733164)

## Test Overview

The TestRig is a tool to automatically run managed code tests and save the test results in a test receipt. The Test Rig will automatically compile the TinyCLR and managed code and load it into the hardware. After loading code the hardware will be reset and the test run.

The user will be able to control the test with a test script if desired. Test data can be collected with the COM port or a logic analyzer. Analysis can be done with an executable (launched by the test script) or Matlab. Test results can be obtained by sending them over the COM port, saving them to a file and having the test script read them, or having them be set by your Matlab script.

## Test Components

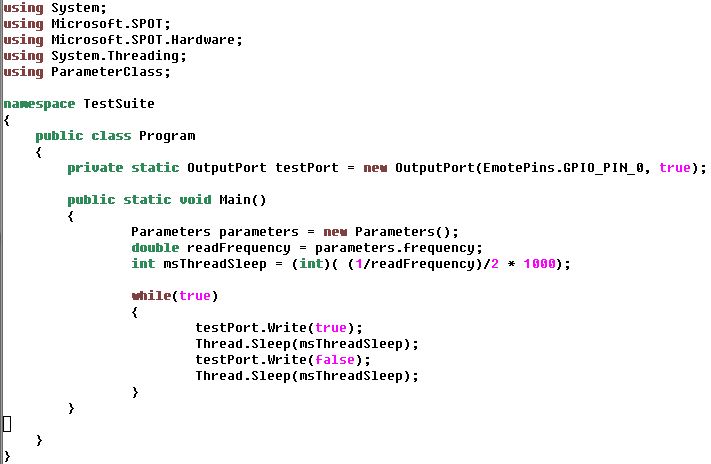
The following are the components that make up a managed code test:

* Test code
* Test description file
* Test parameters
* Test script
* Test analysis script
* Test hookup file
* Test event definition file

## Test Code

The test code is C# code that will run on the TinyCLR microframework. A template C# test can be found in your TestSys directory at: “TestSys\tools\testCreate\TemplateFilesCSharp”.

Your test code can be placed in “Program.cs” starting in the function “Main()”. An example test that toggles a simple GPIO output line is shown here and in “TestSys\GPIO\Src\C#”:



## Test Description File

Every test must contain a test description file and be named, “tests.xml”. A template test description file can be found in your TestSys directory at: “TestSys\tools\testCreate\TemplateFilesCSharp”. This file informs the test tool of necessary compilation and path information in order to run your test properly.

Many of these fields can be left blank and will be populated by the TestRig tool. The items in **bold** must be populated.

The fields for the test description file are as follows:

* Name – name of the test
* Type – either ‘C#’ or ‘Native’
* Description – whatever you want; can be used to help filter your test list when selecting your tests to be run
* TestPath – the path to the directory of the test being written with the root directory being set within the TestRig at “Test Source Path”
* TestProjName – name of your \*.csproj project file
* TesterName – name of person queuing test (will automatically be populated by TestRig tool if left blank)
* TestLocation – name of the location queuing up the test (will automatically be populated by TestRig if left blank)
* TestMFVersion – version of Micro Framework to compile and use
* TestGitOption – can be used to override the user selection in the TestRig; (Use local code; Use archive code; Use archive branch code)
* TestGitBranch – can be used to override the user selection in the TestRig; name of GitHub branch
* TestUsePrecompiledBinary – instead of compiling the Micro Framework source file a precompiled binary can be specified here; the precompiled \*.axf binary needs to be placed in your test directory and the full file name written here; leave blank if not needed
* TestHardware – not currently used
* TestSolution - can be used to override the user selection in the TestRig; (STM32F10x; EmoteDotNow; SOC8200; SOC\_ADAPT)
* TestMemoryType - can be used to override the user selection in the TestRig; (FLASH; RAM; External FLASH)
* TestSolutionType - can be used to override the user selection in the TestRig; (TinyCLR; TinyBooter)
* TestGCCVersion - can be used to override the user selection in the TestRig; (GCC4.2)
* TestSupporting – this parameter is used to load a second device if needed.
  + “load indentical: 2” will load the same binary on two .NOW devices
  + “load support projects: <path>\tests.xml” loads the support binary on the 2nd .NOW
* TestJTAGHarness – can be used to override the programming JTAG device
* TestPowerAutomateSelected – can be used to override whether or not the Adapt test power can be automated on programming
* TestBuild – can be used to override the building of Release or Debug

A sample test description file follows:



## Test Parameters

The test parameter file contains parameters that are used by your test code, the TestRig tool, and your test analysis scripts. The proper way to include your “Parameters.cs” file is shown in the Test Code example shown previously and in the template C# test.

The test parameters allow the user to either use the logic analyzer code built into the TestRig, use the COM port, and to use custom executables to generate test data.

A test timeout must be provided by setting the parameter “testTimeout” to the number of milliseconds the test will timeout in.

To enable the logic analyzer code, set the parameter “useLogic” to “normal” or “I2C” and provide appropriate values to the “sampleTimeMs” and “sampleFrequency” variables. The “normal” setting will save all logic data to a comma separated file. The “I2C” setting will automatically parse logic data into I2C data packets and save that data to a file “testTemp\testData.txt”. The “normal” logic analyzer data is saved to file “testTemp\testData.csv”.

To enable the COM port set “useCOMPort” to “true” and provide appropriate values to “COMParameters”. The TestRig will use the COM port set in the “Test Machine Paths” tab unless the user overrides this selection by setting the “forceCOM” variable to “COM<x>”.

A customized test script can be invoked to do a number of test functions. To enable the script set “useTestScript” to “true”, set the script name in “testScriptName” and then set a timeout in milliseconds with the parameter “testScriptTimeoutMs”. The commands used by the test script will be described in a later section of this document.

All data generated during the test is to be saved in a directory that will be temporarily created during the test: “<C# project directory>\testTemp”. Any data generated by a custom executable is to be placed within this directory which will be deleted after the test is complete.

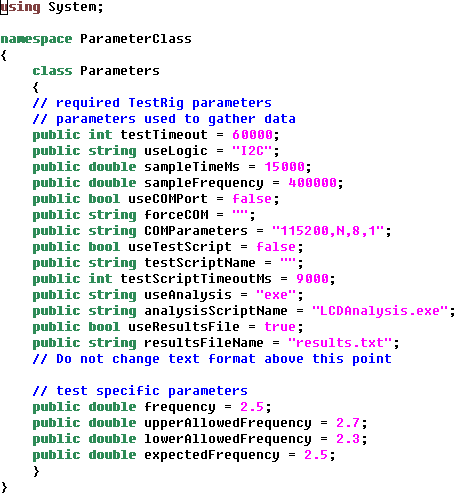
After the logic analyzer and / or test script are finished an analysis script will be run if desired. Make sure the parameter “useAnalysis” is set to “none”, “exe”, or “Matlab” and the name of your script is set at the “analysisScriptName” parameter. The Matlab script is passed the parameter: “dataFileName” which is set to the logic analyzer data file. The analysis executable is passed the data file path and a path for a results file with the name taken from the Parameters class and placed within the testTemp directory.

Any results file generated by your analysis programs can be read in by setting “useResultsFile” to true and setting the result file name to “resultsFileName”.

Finally a test can be delayed before running by setting “testDelay”. This can be used for long-term tests where data does not need to be collected until after a long period oftime.

Parameters to be used by your test script and test program are to be set under the “// test specific parameters” comment.

An example “Parameter.cs” file follows:

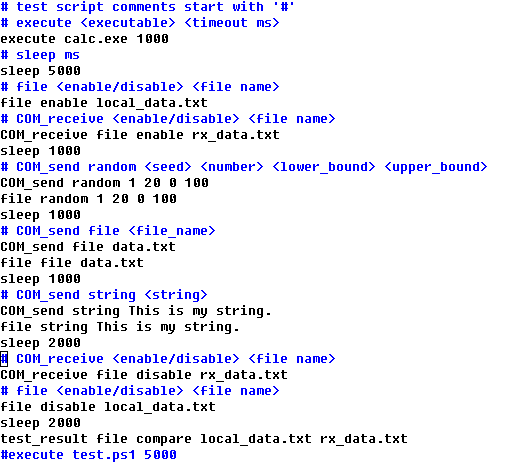


## Test Script

The test script has a number of commands that can be used to generate test data. The script will be run after the Unit Under Test is programmed and then reset. Each line of the test script will be executed in the order it appears in the file until there is no other commands.

* Comments begin with the ‘#’ character and are not executed
* Execute command
  + execute <executable name> <timeout>
    - Example: execute calc.exe 1000 (runs calculator for one second)
* Sleep command
  + sleep <sleep time in ms>
    - Example: sleep 5000 (sleeps 5 seconds)
* Save data to file
  + This command will save generated data to a file that can be used later to compare against a file that is generated by your test. For example you can generate data to a file and then send it across the COM port and verify that received data (saved by COM\_receive command) is the same as that sent.
  + Enable or disable file writing
    - file <enable/disable> <File name>
      * Example: file enable fileSave.txt (saves file data to file: “fileSave.txt”)
  + Save random data to a file
  + file random <seed> <# of numbers sent> <lower bound> <upper bound>
    - * Example: file random none 20 0 100 (saves to file 20 numbers from 0 to 100)
  + Save file data to a file
    - file file <file name>
      * Example: file file data.txt (saves to file the data in file “data.txt”)
  + Save a string to a file
    - file string <Text string>
      * Example: file string This is my string.
* Send data over COM port
  + Random data
    - COM\_send random <seed> <# of numbers sent> <lower bound> <upper bound>
      * Example: COM\_send random none 20 0 100 (sends 20 numbers from 0 to 100)
  + File
    - COM\_send file <file name>
      * Example: COM\_send file data.txt (sends data.txt file)
* String
  + - COM\_send string <Text string>
      * Example: COM\_send string This is my string.
* Receive data over COM port and save it to a file
  + COM\_receive file <enable/disable> <File name>
    - Example: COM\_receive file enable rx\_data.txt (Starts saving data to data file: rx\_data.txt)
* Test results
  + Sets the test results to the test receipt
  + test\_result file compare <File 1> <File 2>
    - Test receipt will be set based on result of file compare
    - Example: test\_result file compare fileSave.txt data.txt
  + test\_result results <file name>
    - Reads in test results from <file name> and sets test receipt parameters
    - Example: test\_result results results.txt

An example test script follows:



## Test Analysis Script

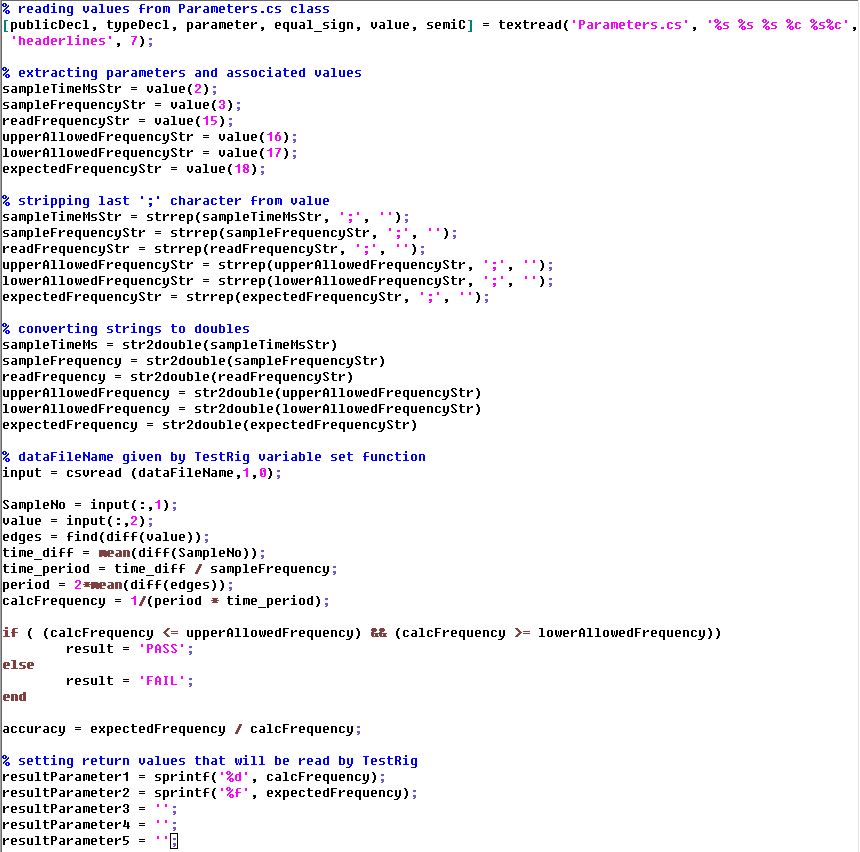
The test analysis script will analyze any collected data and provide a Pass/Fail result, an accuracy result, and some general test feedback to be saved in the test receipt.

The parameters that were passed to both the test code and the TestRig tool need to parsed by the analysis script and used to determine whether the test passed or failed. The test data must be placed within the “testTemp” directory. The logic analyzer data will be automatically stored in a file called “testTemp/testData.csv”. This filename will also automatically be given to the Matlab variable “dataFileName”.

At the conclusion of the test analysis, seven variables must be set. These variables will be saved in the test receipt. The variables to be set are:

* result – a ‘string’ variable that must be set to either ‘PASS’ or ‘FAIL’
* accuracy – a ‘double’ variable that contains a test writer determined value
* resultParameter1 through resultParameter5 – five ‘string’ variables that can contain whatever data the test writer would like included in the test receipt

An example Matlab script that parses ‘double’ variable parameters and calculates the toggle frequency of one of the GPIO pins follows:



## Test Results

There are four ways to get the test results to the TestRig: send them over the COM port, save them to a file and then have the Test Script read them in, have Matlab set them for you, or generate a file with your analysis script and then read it in with the “useResultsFile” parameter. The test will be completed after the final test result (i.e. after resultParameter5 is sent across COM or read from the file) is obtained or the test times out.

* **Send over COM port**

If you enable the COM port in the parameters file you can send test results over the COM port once your test is complete. The TestRig will analyze the data you send it looking for these test results. An example of what you can send follows:

Debug.Print("result = PASS\r\n");

Debug.Print("accuracy = 1.2\r\n");

Debug.Print("resultParameter1 = p1 return\r\n");

Debug.Print("resultParameter2 = p2 return\r\n");

Debug.Print("resultParameter3 = p3 return\r\n");

Debug.Print("resultParameter4 = p4 return\r\n");

Debug.Print("resultParameter5 = p5 return\r\n");

* **Save to file**

Your custom executable invoked by the Test Script can save test results to a file. This file can then be read by the Test Script with the command: test\_result results <results file>. This file format is also used by the “useResultsFile” parameter. The correct format of the file is as follows:

result = PASS

accuracy = 1.2

resultParameter1 = p1 return

resultParameter2 = p2 return

resultParameter3 = p3 return

resultParameter4 = p4 return

resultParameter5 = p5 return

* **Matlab sets the test results**

Matlab can set the test result as seen by the example Matlab script shown earlier in this document.

* Read analysis script results

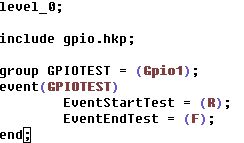
## Test Hookup File

The test hookup file describes what pins will be sampled by the TestRig tool. An example of a hookup file that saves data from GPIO pin 1 follows. The file will have the same root name as the project except it will end in \*.hkp instead of \*.csproj.



## Test Event Definition File

The test event definition file describes what data events will be saved. An example of an event definition that will save all data line transitions on a GPIO pin follows. The event definition file will have the same root name as the project except it will end in \*.edf instead of \*.csproj.



## Writing Great Tests

There is an art to writing tests to be sure. Written incorrectly, it can easily give false assurances.

During the development of the test, the test writer needs to purposely break the code (generally by placing an infinite loop in tinyhal.cpp) and execute the test to verify that it does indeed fail.

Also, it is up to the designer to choose a proper test duration. For example, virtual timer tests all initially ran correctly passed, but they were not executed for very long periods of time. A test was written to catch this type of error, but if a problem takes hours to manifest and a number of tests take six hours each or more to run, then a user could run tests all weekend and not finish. I'm working on a way to filter out and run separately tests that take many hours to run, as I think that will interfere with the development process if engineers can't test all major components in a set of tests overnight.

It is up to the designer to properly identify test condition, data, and duration for all features. For example, if tests pass sometimes, but not always, this indicates a problem is not tested thoroughly enough to definitively give a valid pass. This is where the art of writing the tests intersects with ensuring good coverage.

Test development is an ongoing iterative process. If a new bug is found and fixed, a new test should be written to test for this bug. This test should fail when the bug is present and pass when fixed in the code.

## Further Reading

Grenning, James. *Test Driven Development for Embedded C*. Ed. Jacquelyn Carter. Raleigh: The Pragmatic Bookshelf, 2011. Print.

Wikipedia contributors. "Necessity and sufficiency." *Wikipedia, The Free Encyclopedia.* Wikipedia, The Free Encyclopedia, 23 Jan. 2015. Web. 3 Feb. 2015.

Wikipedia contributors. "Verification and validation." *Wikipedia, The Free Encyclopedia*. Wikipedia, The Free Encyclopedia, 27 Jan. 2015. Web. 3 Feb. 2015.

## Glossary

Until we develop a common language about testing within Samraksh, here are some variations of acronyms used in Test System and Test Suite.

MUT – mote under test

DUT – device under test

UUT – unit under test

DAC – digital analog converter

PRNG – pseudo random number generator

GPIO – general purpose input output

Verification – conforms to a specification

Validation – meets the user’s needs

Necessary test – test recognizes true positive

Sufficient test – test guarantees against false positives

Necessary and sufficient – the test passes if and only if the subject is valid