Embedded Unit Testing Framework

Erik Bergersen

Master of Science in Computer Science

Submission date:	June 2013
Supervisor:	Tor Stålhane, IDI
Co-supervisor:	Øyvind Grotmol, Energy Micro

Norwegian University of Science and Technology
Department of Computer and Information Science

Problem Description

The aim of this project is to make it easy to do thorough unit testing of C/C++ code written for the EFM32 microcontrollers, by creating a testing framework. The core problem is that the embedded code interacts with a number of peripherals on the microcontroller, which are not available when running the tests. Thus, it is essential to create mock objects or stub functions for all EFM32 modules, which allows the test instrumentation code to control exactly which response is given to the code-under-test (CUT), and also to capture the messages sent by the CUT in order to compare it with the expected behavior.

The project will see if unit testing can reduce the number of errors before release of software, and that the work-hours used on testing can be reduced. Show that Unity and CMock is suitable as a framework for unit testing embedded software. The project will also look at problems and barriers that prevent developers from using unit testing during development of embedded software.

Abstract

This thesis addresses the challenges with unit testing of embedded software. Embedded software uses peripheral devices that are not available during testing, which results in higher barriers to use unit testing, compared to normal software. But even with these barriers there are no problems with unit testing of embedded software.

The study looks at challenges with unit testing of Energy Micros software library, and solutions to the problems with automated testing of embedded software. Use of automated testing will reduce the number of errors when code is refactored. Use of unit testing will also make it possible to take advantage of agile methods like test-driven development. The main goals for the study is to see if CMock is suitable for generation of mock modules and if testing with mock modules can reduce the number of errors in software.

Unity is a test harness that supports the needed assertions for testing of embedded software. CMock is able to automatic generate mock modules, but has a few problems with inline functions and custom types, but it is possible to solve these problems. Even with these problems CMock is able to generate mock modules that is working, and supports the testing of embedded software.

An experiment was performed to see if testing of embedded software with mock modules could reduce the number of errors. The result was significant and showed that the number of errors was reduced by using mock modules, compared development without testing. Observations were also done during the experiment, which found some problems with use of mock modules during testing. The experiment was done by undergraduate students in computer science.

Use of unit test and mock modules as a substitute for hardware during automated testing on a native computer, it is possible to reduce the numbers of errors and refactor code without breaking the existing functionality.

Sammendrag

Denne rapporten tar for seg utfordringene med enhetstesting av embedded programvare. Embedded programvare bruker perifere enheter som ikke er tilgjengelige under testing, dette resulterer i høyere barrierer for å bruke enhetstesting, sammenlignet med vanlig programvare. Men selv med disse barrierene er det ingen problemer med enhetstesting av embedded programvare.

Studien ser på utfordringer knyttet til enhetstesting av Energy Micros programvarebibliotek, og løsninger på problemer med automatisk testing av embedded programvare. Bruk av automatisert testing vil redusere antall feil etter refaktorering av kildekoden. Bruk av enhetstesting vil også gjøre det mulig å ta i bruk smidig utvikling, som for eksempel testdrevet utvikling. Hovedmålene for studien er å se om CMock er egnet som et rammeverk for generering av mock moduler og om testing med mock moduler kan redusere antall feil i programvaren.

Unity er et testrammeverk som støtter de nødvendige sammenligner under testing av embedded programvare. CMock er i stand til å automatisk generere mock moduler, men har problemer med ‘’inline” funksjoner og håndtering av strukturer og andre ikke primitive datatyper, men det er mulig å løse disse problemene. Sett bort fra disse problemene er CMock i stand til å generere mock moduler som fungere og kan støtte testingen av embedded programvare.

Et eksperiment ble utført for å se om testing av embedded programvare med mock moduler kunne redusere antall feil. Resultatet av eksperimentet var statistisk signifikant og viste at antall feil vil bli redusert ved bruk av mock moduler, sammenlignet med utvikling uten testing. Observasjon ble også gjort under forsøket, dette oppdaget noen problemer ved bruk av mock moduler under testing. Eksperimentet ble utført av studenter ved datateknikk.

Ved å bruke enhetstester og mock moduler når hardware ikke er tilgjengelig for automatisk testing på vanlig datamaskin, er det mulig å redusere antall feil og refaktorere kode uten å ødelegge eksisterende funksjonalitet i programvaren.

PREFACE

The report was written as final thesis for the master degree in computer science at Norwegian University of Science and Technology (NTNU). The thesis was carried out in the spring of 2013 and looked at unit testing for embedded software.

The aim for the project is to do easy unit testing of C/C++ code, which are written for EFM microcontroller, by studying Unity and CMock as unit test framework for embedded software. The project was given by Øyvind Grotmol at Energy Micro.

I would like to thank my supervisor Tor Stålhane for his continuous input and guidance during the project. I would also like to thank Øyvind Grotmol and Marius Grannæs at Energy Micro for the help during the project. I appreciate that XCOM14 could get participants to my experiment.

Trondheim, June 10, 2013

Erik Bergersen

CONTENTS

Preface **i**

Contents **iii**

List of Figures **vii**

List of Tables **ix**

1 Introduction **1**

1.1 Problem Definition 1

1.2 Motivation 1

1.3 Goals 2

1.4 Approach 2

1.5 Outline 3

2 Background **5**

2.1 EFM32 5

2.2 Unit Testing 9

2.3 Testing of Embedded Systems 12

2.4 Existing Code 16

3 Embedded Unit Test Framework **19**

3.1 Introduction 19

3.2 Dependencies 20

3.3 Software 22

3.4 Testing of Peripherals 25

3.5 Use of Mock Modules 32

3.6 Testing of Examples . . . 34
3.7 Testing of Drivers . . . 41
3.8 Automated Build System . . . 42
3.9 Summary . . . 45

4 Experiment **47**
4.1 Definition . . . 47
4.2 Planning . . . 49
4.3 Operation . . . 52
4.4 Analysis & Interpretation . . . 54
4.5 Summary . . . 63

5 Discussion **65**
5.1 Experiment . . . 65
5.2 Mock Modules . . . 66
5.3 Embedded Testing . . . 68
5.4 Test Strategy . . . 69
5.5 Agile Development and Test-driven Development . . . 70
5.6 Regression Testing . . . 71

6 Conclusion **73**
6.1 Conclusion . . . 73
6.2 Further Work . . . 74

Bibliography **75**

Abbreviations **77**

A Experiment Guidelines **79**
A.1 Introduction . . . 79
A.2 Instruction . . . 79
A.3 Tasks . . . 83
A.4 When Finished . . . 91

B Experiment Test Cases **93**
B.1 Task 1 . . . 93
B.2 Task 2 . . . 95
B.3 Task 3 . . . 97
B.4 Task 4 . . . 99

B.5 Task 5 101

C Mock Preperation Script **103**

C.1 Script 103

C.2 Config-file 106

LIST OF FIGURES

2.1 Bit-banding . . . 7
2.2 Software Layers . . . 7
2.3 TDD State Machine [16] . . . 11

3.1 Blink Dependecies . . . 21
3.2 Blink Dependecies with Mock Modules . . . 21
3.3 STK3700 Hardware Layout [12] . . . 35

4.1 Boxplot of Test Results . . . 55
4.2 Distribution of Errors . . . 55

A.1 Connect with energyAware Commander . . . 81
A.2 Upload with energyAware Commander . . . 81
A.3 STK3700 Hardware Layout [12] . . . 82
A.4 Text Pattern . . . 88

LIST OF TABLES

4.1 Errors per Participant 54
4.2 Errors per Test Case 56
4.3 T-test Results 62

B.1 Test case TC1-1 93
B.2 Test case TC1-2 93
B.3 Test case TC1-3 94
B.4 Test case TC1-4 94
B.5 Test case TC1-5 94
B.6 Test case TC2-1 95
B.7 Test case TC2-2 95
B.8 Test case TC2-3 96
B.9 Test case TC2-4 96
B.10 Test case TC2-5 96
B.11 Test case TC2-6 97
B.12 Test case TC3-1 97
B.13 Test case TC3-2 98
B.14 Test case TC3-3 98
B.15 Test case TC3-4 98
B.16 Test case TC3-5 99
B.17 Test case TC4-1 99
B.18 Test case TC4-2 100
B.19 Test case TC4-3 100
B.20 Test case TC4-4 100
B.21 Test case TC5-1 101

B.22 Test case TC5-2 101
B.23 Test case TC5-3 102
B.24 Test case TC5-4 102
B.25 Test case TC5-5 102

CHAPTER 1 INTRODUCTION

This chapter is an introduction to the thesis. The chapter contains; problem definition, motivation, context and an outline of the report.

1.1 Problem Definition

The task was given by Øyvind Grotmol in Energy Micro. The project's aim is to look at an embedded unit test framework, which make it easy to do unit testing of C/C++ code. This code is written for Energy Micros microcontrollers. The company needs a unit test framework to take advantage of agile methods during development of example projects for Energy Micros development kits, and development of the software library for the microcontrollers.

One problem with unit testing in the embedded space is that the code interacts with peripherals on the microcontroller that not can be accessed during tests. To be able to test this logic there is a need to create stub functions and mock objects that make it possible to test instrumentation code to check which response is given during the test, and to capture the messages in order to compare it with the expected behavior.

The thesis will look at different layers of the software library that Energy Micro provides, with focus on the examples that are provided for the different development boards.

1.2 Motivation

The motivation for solving this problem is that Energy Micro has more than 200 example projects, which is available for the users of theirs starter and development kits. Energy Micro has nine different development boards. Many of the examples support several of these development boards, in addition they have to support four different IDEs (Integrated Development Environment). The problem with testing is caused by the development boards and more than 200 example projects. In total this will be more than thousand binaries, which is a huge amount of binaries to test if changes are made in the software libraries. Testing can be supported with unit tests that are run automatically, instead of manual system tests, which would be very expensive.

1.2.1 Purpose

The purpose of the thesis is to find methods that can be used to reduce the number of errors that are created during refactoring and modification of code. By using unit testing it is also possible to adopt agile methods like test-driven development. If more of the tests activities can be automated, time used on testing can be reduced.

Use of mock modules/objects will be essential in thesis, since interaction with hardware is not possible during running of tests on a host computer.

The objectives of the thesis is to verify that Unity and CMock is suitable as a unit test framework for embedded systems and to see if unit testing can work as a method for regression testing of a software library for several different microcontrollers.

1.3 Goals

For the project there are both research goals and goals that Energy Micro has for testing their library with unit testing.

The research goals for the project are as follow:

RQ1 Is it possible to reduce the number of errors during development of new features, modification and refactoring of existing functionality by using mock modules to verify the desired behavior of the system?

RQ2 Show that CMock is a suitable framework to create mock objects, which can simulate the behavior of the hardware, which is not available during testing on a native system.

The goals for Energy Micro are:

1. Reduce the number of errors in the code, before releasing it to the customers.
2. Take advantage of agile development, with unit testing and test-driven development.
3. Be able to refactor or change the code on a later time, without breaking the code.

The first of the goals is connected with RQ1, since they both look at the reduction of errors. The second goal is accomplished if Unity and CMock is suitable for testing. The last goal is fulfilled if the framework can be used to create tests, which can be run automatic on a host computer, for the code that is changed.

1.4 Approach

The thesis consists of two parts, where the first is to use the test framework in practice to test functionality in the software library. The second part will be an experiment where we look at how mock modules can reduce the number of errors during development.

To be able to use a unit test framework to perform regression testing on a software library that supports about a hundred different microcontrollers, we need to look at advantages and disadvantages of this solution. Testing of peripherals, kit specific code and example projects will be done to find barriers and challenges with unit testing, in order to find a solution. When testing a software library for microcontrollers, it is impossible to test all the code, since it will depend on hardware. The focus will be on the mock modules created by CMock, since this is an important tool to be able to solve the tasks, when hardware is not available under testing.

The experiment will be used to study the reduction of errors when the developers use unit tests based on mock modules to add new features to the system, and under modification of existing features. The participants in the experiment will implement features in a few tasks that are based on examples for the Energy Micro Giant Gecko STK3700 Starter Kit.

1.5 Outline

Background gives background information about hardware, software and techniques that is used in the thesis.

Embedded Unit Test Framework summarizes testing of the Energy Micro software libraries, and looks at challenges and solutions by using unit testing to perform regression testing.

Experiment contains the experiment that was performed during the thesis, which includes the definition, planning, operation, analysis & interpretation and conclusion.

Discussion contains the discussion of the results from the testing and the experiment, and a comparison of these two parts.

Conclusion contains the conclusion, together with further work.

Appendix contains guidelines for the experiment, and the test case used for measurement in the experiment. It also contains source code, tests and scripts as documentation when needed as a reference for other parts of the report.

CHAPTER 2 BACKGROUND

This chapter describes some of the hardware features that are available on the EFM32 microcontroller. The chapter also includes some unit testing theory, how testing is performed on embedded systems/software and a part on how to write tests for code without tests.

2.1 EFM32

This section gives a short introduction to the EFM32 microcontroller family developed by Energy Micro, the ARM Cortex-M processor and a short description of a few peripherals that are included on the microcontroller. It also includes a short description of the development boards that are produced by Energy Micro, the support for IDEs and description of the parts in the software library that are provided for development on the microcontrollers.

2.1.1 Energy Micro

Energy Micro is a Norwegian company that was founded in 2007. Energy Micro develops the EFR (Energy Friendly Radios) family and EFM (Energy Friendly Microcontrollers) based on ARM Cortex processors [5]. The focus for the company is to provide microcontrollers with low energy usage. This project will focus on the embedded unit testing of the software for the microcontrollers.

2.1.2 Hardware

The main part of the hardware in this project is the microcontroller, which is based on an ARM processor. In addition there exists several peripheral devices, and hardware features that are available on the development boards.

Microcontroller

A microcontroller consists of an on-board memory unit, a microprocessor and I/O devices, and it is designed for low-cost systems [24]. The EFM32 microcontrollers are delivered with different amount of RAM and flash memory, different peripherals and different packages. All EFM32 microcontrollers use the ARM Cortex-M3 microprocessor.

ARM Cortex-M3

The Cortex-M processor is the processor that is used in the EFM microcontrollers. The Cortex-M processor architecture is created by ARM, and is made for ”tomorrows embedded applications” [1]. It is designed to be energy efficient, and uses a processor built on the RISC architecture.

Peripherals

The EFM32 microcontrollers have several peripheral devices. The microcontrollers use the peripherals for input, output and storage. The peripherals on the microcontroller are designed for energy efficiency. Below is a list of some of the peripheral device that is available on the EFM32 microcontrollers:

ACMP The Analog Comparator (ACMP) is a module that compares analog signals and gives a digital voltage to indicate which of the analog signal that is highest.

ADC The Analog to Digital Converter (ADC) is used to convert analog signal to a digital signal. The peripheral can give a digital output with a resolution of 12 bits. The input can be selected from 6 internal signals and 8 external pins [11].

AES Advanced Encryption Standard (AES) is a peripheral for encryption and decrypting. The module supports 128 and 256-bits encryption [11]. The hardware module uses considerable less energy, than what a software implementation would use.

UART Universal Asynchronous Receiver/Transmitter (UART) is a serial IO peripheral, which support full duplex and half duplex communication.

LCD This peripheral is used to drive the segmented Liquid Crystal Display (LCD). The LCD can for example be used to display text.

Watchdog The purpose of the watchdog timer is to reset the system to a known state if it encounters a system failure. The watchdog has a timer with a configurable period that is the timeout threshold. The timer will count continuously with the signals given from an oscillator. The processor must feed the watchdog with a reset signal before timeout.

Bit-banding

Bit-banding is a technique that is used to map words to bits. This is done since writing to a single bit will require reading the word, modify and then write the bit back. By using bit-banding, it is possible to do this in a single operation [11, p. 16]. Figure 2.1 illustrates bit-banding.

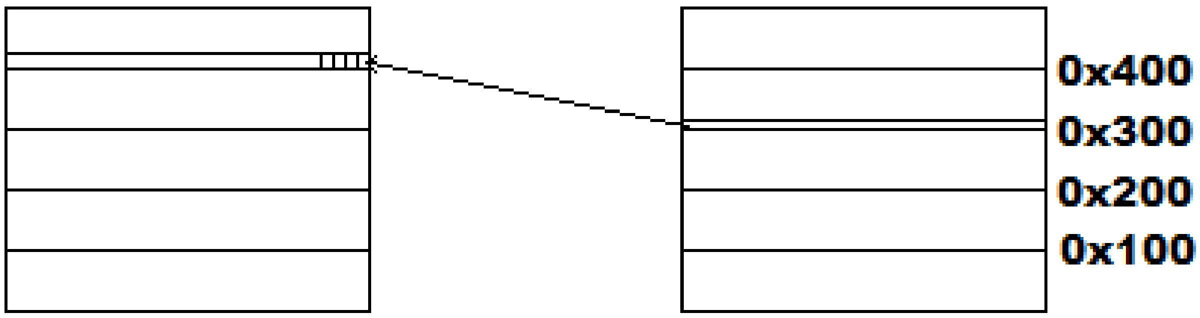


Figure 2.1: Bit-banding

2.1.3 Software libray

The software library that is delivered by Energy Micro consists of several layers as seen in Figure 2.2. Each of these layers is described below.

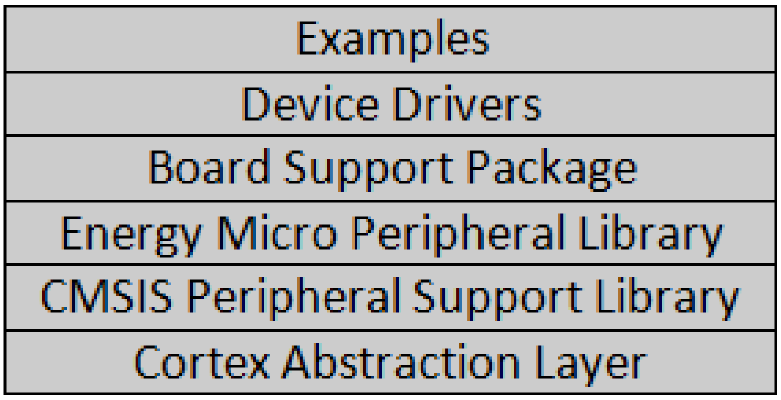


Figure 2.2: Software Layers

Cortex abstraction layer CMSIS (Cortex Microcontroller Software Interface Standard) is a hardware abstraction layer for the ARM Cortex-M processors. This layer is independent of the microcontroller and provides an interface to the ARM processor architecture.

CMSIS peripheral support library This layer consists mainly of memory registers definitions and bit field definitions. The layer does not include any functionality that can be tested. Each microcontroller has different specifications in this layer, depending on the availability of memory and peripheral devices on the microcontroller. The support layer also includes a system layer for the microcontroller on the ARM Cortex-M3.

Energy Micro peripheral library This layer is a HAL (Hardware Abstraction Layer) and an API (Application Programming Interface) for each of the peripherals, and has little logic. The peripherals will be tested later in this thesis, to look at possibilities and problems with testing on this layer. This layer is used by every microcontroller, where the CMSIS peripheral support library defines the ability for the given microcontroller. Although this layer has little logic, unit tests will be needed, because these are necessary for regression testing, and for testing with test-driven development.

Board support package This is a support package for the starter/development kit, which includes support functions and communication package definitions for the kit. The board support package also includes drivers and examples that are described below. Each starter/development kit has its own board support package. It may be necessary to unit test some of the functionality of the board support functions.

Device drivers The drivers are used to demonstrate functionality on the development board. The device drivers are created for several starter and development kit, and configuration is used to specify them for each kit. Examples of drivers are segmented display driver and NAND flash driver. Unit testing of the device drivers is necessary, and it can be tested on the development board, or with use of mock objects.

Examples The examples are given as a quick start to programming on the development boards. The examples demonstrate functionality that the development board has. Each of the kits has 10 - 50 different examples. Some of the examples exist on several of the kits, but they are different due to hardware and memory differences. Most of the examples support four IDEs. Some of the examples will be tested later in the thesis.

2.1.4 Integrated development environment

Energy Micro support four IDEs (Integrated Development Environment) [12]. The support gives the ability to open example projects in the given IDE. The IDEs that are supported are:

- IAR Embedded Workbench for ARM
- Rowley Associates - CrossWorks for ARM
- CodeSourcery - Sourcery G++
- Keil - MDK-ARM

When compiling and building, each of the IDEs will create a binary, which will result in four binaries for an example. This raises the complexity with testing due to the amount of binaries that will need testing. Compiling each example for all the IDEs is necessary, since the IDEs uses different compilers. The C programming language standard specifies the language, but it does not specify behavior in all cases, the language has cases of undefined behavior[1], implementation-defined behavior[2] and unspecified behavior[3]. It is thus necessary to compile and test on all IDEs/compilers.

[1]behavior, upon use of a nonportable or erroneous program construct or of erroneous data, for which this International Standard imposes no requirements [17, 3.4.3]

[2]Unspecified behavior where each implementation documents how the choice is made [17, 3.4.1]

[3]use of an unspecified value, or other behavior where this International Standard provides two or more possibilities and imposes no further requirements on which is chosen in any instance [17, 3.4.4]

2.1.5 Development boards

Energy Micro has several kits that can be used by developers to test and debug the EFM microcontrollers. Energy Micro provides two types of kits; development kits and starter kits. The starter kits are simple kits that can be used to try out some of the features in the microcontrollers. These kits are mostly an evaluation of the microcontroller. The developer kits are made for advanced users that want to develop embedded systems. The developer consists normally of more functionality and more advanced features than the starter kits, and has better support to build custom circuits. With the advanced energy monitor, the users are able to see the energy usage on the system, and are able to evaluate and debug with the developer kit. With tools to monitor energy usage, the developers are able to optimize their system, to use as little energy as possible. The starter and developer kits differ for each microcontroller family developed by Energy Micro.

2.2 Unit Testing

This section gives an introduction to unit testing, description of a few test doubles and a description of test-driven development.

2.2.1 Unit testing

Unit testing is a technique for white box testing, where the testers now the internal structure of the system. Unit testing is about breaking the code down to small units of code, a unit could be a single method in the code or a complete class (object-oriented programming). Test cases are written to verify the correct behavior of a unit, and the test cases are independent of each other. To be able to do unit testing, the developers will normally need a test harness, which will include functionality to do assertions, setup for test and afterwards clean-up after the test. In addition they normally have some functionality for test report.

When unit testing, some part of the system may not be available, because it needs hardware that is not available during testing, functionality that not are fully developed yet or using functionality like network or databases that will result in that tests uses too long time to finish. For these parts it could be necessary to use mock objects, test stubs or other test doubles to simulate and be able to run the tests. The benefits of using unit tests are that it is possible to test and find errors early in the process, and it can be used to check that code is not broken during changes or refactoring of the code. Unit tests itself is documentation that the code is tested and should behave like the developers intended.

Using unit testing during development on embedded system will reduce the number of times it is necessary to compile the code for the hardware, upload it and then run the tests on the hardware. During testing with unit tests it is important that the code is testable, by structuring the code so only one layer do access the hardware, which will improve the design of the software. When a problem is identified it is easier to fix errors that are discovered by the tests, than having to debug on the hardware. This will reduce the amount of work-hours the developers use. During development of hardware that is not ready for production, it is possible to start implementation of software and test it using unit test and mock objects, before the hardware is ready, which mean that development can start early in the process.

2.2.2 Test doubles

During testing of code, several collaborators are working together, like methods, classes, data and modules. The unit that is tested with a unit test depends on several of these collaborators. A test double represents and acts like a method, class, data or module [16]. There exist many kinds of test doubles; this part will only look at stubs, fakes and mock objects.

Stubs

A stub is a function that is created to only return a value, without any functionality. Similar to a test dummy, that is a function that only is created to be able to compile/link an application [16].

Fakes

A fake object is an object that is used to break dependencies during testing. A fake is a collaborator that acts as an object or a module during testing [15]. The fake has implemented some functionality, but is not fully implemented, since the main purpose is to break the dependency. The fake can be used to break the dependency to for example hardware, which will make it possible to run the test without hardware. The fake will never be able to detect any errors, but will act as a collaborator that may simulate some form of behavior.

Mock objects

A mock object is an advanced version of a fake module/object. The difference is that the mock object/module uses internal assertions [15]. When using the mock objects, the developers sets the expectations and inject return values before the unit tests are run, afterwards the expectations are verified. The verification in mock object will check that the method is called and will verify the parameters with assertions. The mock module will also verify that the method is called the right number of times. It can also verify that function calls are done in the correct order.

When using unit testing, it is necessary to use objects and modules that can behave like components that are not ready for testing or is not available during automated unit testing, for example hardware. The mock object can be used to describe the call between modules, and the expected parameters and return values [16].

The programming language that is used for the embedded software in this thesis is C. Since C is not an object-oriented language, mock object will be referred to as mock modules instead.

2.2.3 Test-driven development

TDD (Test-driven development) is a software development process. The purpose is to ensure that all code that is written is being tested. The main idea is that you write code if you have a failing test, and a cycle in the development is finished when all the tests are passing. A cycle in test-driven development is commonly known as red-green-refactor [16]. This means that you first write a test that is failing, the reason why it is failing should be a compilation error, because the compiler cannot find the definition of the function the test is calling. The developer should then fix this error by adding a stub function, run the tests to verify that they are green. Then the tests should be changed so it asserts values for different input values. When the code is red again the feature can be fully implemented. When the tests are run again they should be green and the developers are able to refactor the code, if it is needed. When using this TDD cycle, it will be possible to discover and eliminate bugs early. Figure 2.3 [16] display a state machine to describe how test-driven development is used in C programming.

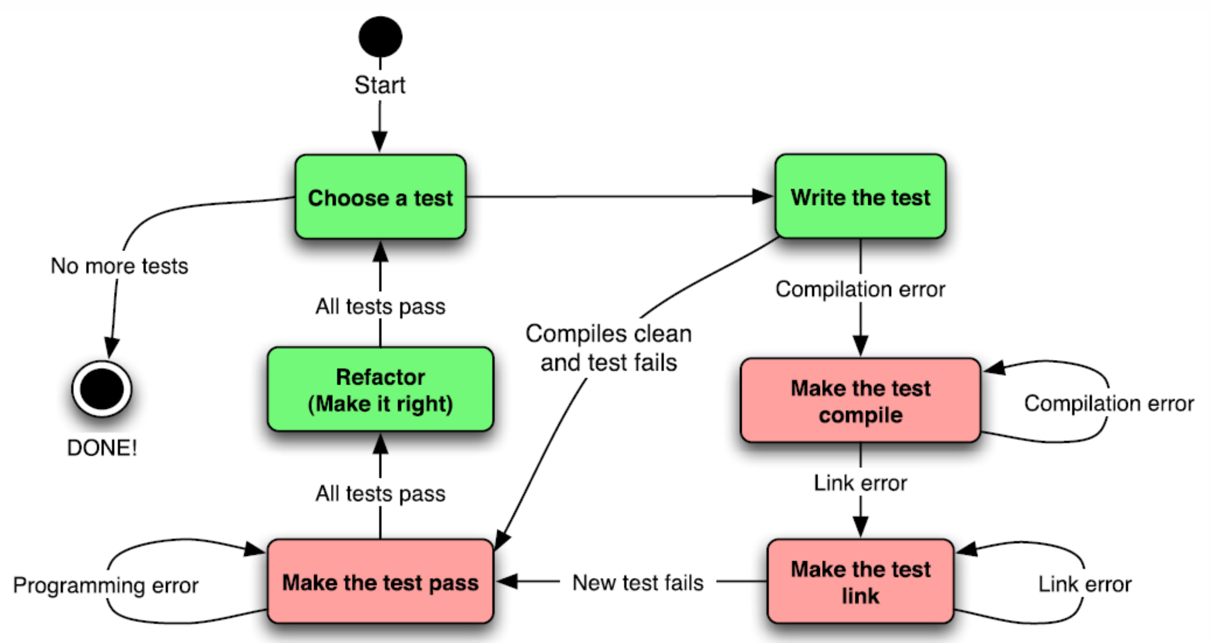


Figure 2.3: TDD State Machine [16]

When using a process like TDD during development on an embedded system, the developers will have to write code that is intended to run on two different systems; the development system and the hardware that will be used for production. With this ready the developers can run tests during development, which will be used to verify that the production code is correct without the need of an embedded device.

2.2.4 TDD for embedded software

It is possible to use test-driven development for embedded systems. Each time new functionality is created, a test will be created for this function. When functionality is refactored, modified or functionality added, the unit tests can check that the functionality that existed before modification is still working afterwards. This can be used as automated regression testing of the software. To ensure that a system with low number of errors are delivered, system tests will still be necessary, but with the automated regression tests, the time that will be needed to use for system and acceptance testing will be reduced.

Since one of goals is to take advantage of agile methods, test-driven development is a good suggestion. TDD can also be used together with other agile methods like Scrum or Extreme Programming.

Test-driven development is a method that will help to ensure that the software is well-tested, which can be used to find errors in the hardware. These errors can be found since well-tested software often are easier to understand, and confidence with that code is well-tested, will prevent the developer from starting extensive debugging, because the developer thinks the error is within the software. In ”Effective Test Driven Development for Embedded Software” [18] there is suggested a four level strategy, the first level is automated unit testing and for each level more human interaction is necessary. The levels described are:

Automated unit testing Unit tests are added during development, and mock objects generated when needed. The mock can be generated in a MCH design pattern to be able to decouple the logic from hardware. The automated unit tests are run by the developer. The tests are run on development boards, emulators or as cross compiled tests on a computer. The tests can also be automatic tested by a continuous integration tool.

Hardware level testing A combination of unit testing and interaction by the tester is used to test user interaction and hardware. The most important part of this testing is the hardware functionality [18]. These tests will be run infrequently compared to automated unit testing.

Communication channel testing Use developer tools to capture test results from the embedded device. Both hardware and software will be tested, and it will require interaction from the testers to create communication events during testing.

End to end system testing This layer executes the scenario tests, which is to test the user scenarios that are defined for the system. Important things to test are timing issues, user interface and the responsiveness of the system.

2.3 Testing of Embedded Systems

This section will briefly look at unit testing on embedded systems, and how system testing can be done for software and hardware.

2.3.1 Hardware testing

Testing of embedded software is in many ways different from regular software for personal computers. Some of the differences are the architecture, platform, processing speed, memory available and storage capacity. Debugging is also different, on software for computers, debugging can be done through the IDE. For embedded systems there will be a need for a development board with a debugging interface, for example JTAG, which can be used for example for single stepping and break pointing when debugging on the embedded device. JTAG was developed as a test interface, but is also widely used as an interface for embedded debugging [22]. When testing embedded software, testing must cover both hardware and software, since errors can exists in both or in a combination of them.

In “Effective test driven development for embedded software” [18], three methods of testing is described: ad-hoc testing, debugging and final testing. The article describes the current state of these and the shortcomings with the three methods, when testing on embedded systems.

Ad-hoc testing and experimentation is often used during development to discover infrequent cases in the system, usually to test system boundaries. The knowledge that is gained from such testing and experimentation during implementation of the functional code will usually be discarded or shelved. So later in the development process this knowledge is lost, and work-hours will be used later during development, due to the fact that tests are not kept updated [18].

Since errors can come from both hardware and software, embedded software uses many debugging and inspection tools during development. There exist many advanced debugging tools for embedded systems. The effect of these tools has however, some bad side effects. The use of debugging tools can result in that the software is designed for debugging and not for testability. Finding errors with debugging is difficult, since it is unknown where the error is. Debugging to find a specific error is something that is done once, it is not necessary to design the code for debugging when the errors is found and corrected. When using debugging widely to test and develop, it will often result in bad coding practices, because the code is not designed to be testable, so hardware and software is not decoupled [18]. A positive effect with debugging is that it can be used to find errors, when developers not able to understand the results from a given input. Another weakness is that test equipment for microcontroller can be expensive, so it will be impossible for low budget projects [14].

Development processes like waterfall are often used during development of embedded software systems. The process starts with a design of the system, then development of the system, testing of the system and finally production. Testing is a costly stage of the development and it is difficult to find the correct point for when the testing is finished and the system is ready for production. Since testing is one of the last stages, testing can be reduced for projects that are delayed and need to meet a deadline. The result will be that the system is not tested well enough and can contain critical errors. When testing is done late in development, there is a chance that the code is not designed and developed for easy testing [18].

2.3.2 Unit testing

There exist several unit test frameworks created for testing of embedded software. There are different ways in which unit tests can be used to test embedded software. “Functionality and Design of the CMock framework” [20] lists three common soultions:

- A development board or a development kit as described in the preliminary study chapter can be used for automated testing of the embedded software. The test binary is deployed on the development board and then executed, the results from the tests are written to a memory location or transferred back to the computer through a debugging interface. Testing on a development board is a slow method since it requires deployment to the development board, and the embedded system usually has a processor that is much slower than a normal computer. The big advantage is that the test is run with access to the peripheral devices.

- An emulator that can be used to simulate the behavior of the hardware. The advantage from the development board is that the emulator is much faster, but like the development board it will need a debugging interface to transfer the results from the tests. A debugging interface is not necessary a feature that is provided by the emulator and the support for peripheral is not always available, or is only partially implemented.
- The last solution is to run the tests on the local computer. This is the fastest solution. But the tests have no access to peripheral devices and real hardware. To be able to test with usage of a peripheral device, there is a need to develop mock objects and stub functions to simulate the behavior, but the results may not be the same as on the embedded device. There might be problems with the compatibility between the embedded device and the local computer, since the platforms are different and uses for example different sizes on data types.

2.3.3 System testing

System testing looks at testing at all the stages in the development of an embedded system.

Hiearchy of testing

"Testing Complex and Embedded Systems" [19] explains a bottom to the top hierarchy of testing software for embedded software. The different levels of testing are described below:

Unit Testing This is testing of the smallest units of software. This level includes build tests and regressions tests, where the build tests is responsible for testing the functionality that was changed, and regression test is run to verify that no other software components was affected by the changes in the code.

Component Testing Is testing on a higher level, where tests are focused on the components instead of the units.

Integration Testing Test the integration of software and hardware, to test interaction and that the systems behavior is as intended.

System Testing This is integration of the subsystems, which includes multiple hardware and software components. Some suggested system level options are: FAT (Factory acceptance testing), OT (Operational testing) and MQT (Manufacturing qualification testing).

Field Testing Testing of software in real environments, the system is complete and is tested as it would be used in real-life.

Simulation and emulation

The goals of simulation are to derive requirements and find unknown interactions and details in the system design [19]. The simulation is often a model of a function or process. The requirements that are derived are used to create the tests that are needed in the development process.

Emulation is different from the simulation, since an emulator is a replica of software/hardware that is used for debugging and testing [19].

Model to final product

Development of embedded software needs many test activities, during the different stages of the process. In ”Testing embedded software” [13] a process is described for building embedded systems in stages. These stages are simulation, prototyping, pre-production and post-development.

When the requirements for a system are finished, a simulation model is created to verify requirements and to optimize the system design. The simulation is usually run in dedicated software tools to simulate the model [13]. The simulation is executed in three stages.

- One-way simulation: Testing of the system in isolation, where only one input is tested at a time. Tools are used to generate input to the model.
- Feedback simulation: Testing the interaction between different parts in the system. Initially the system should be simulated only with needed parts, environments is not part of this simulation.
- Rapid prototyping: Testing of the system with a real environment. This is done by connecting the model to the real environment by connecting the computer to actuators and sensors.

The second stage in the process is to release a unit for pre-production, and validate the model used in simulation. The model is incrementally replaced with real hardware and software. The prototype is run on either the real hardware or with an emulator on a host computer. This stage consists of several test levels; software unit testing, software integration testing, hardware integration testing and environmental testing [13]. For each of the levels the software, processor, hardware and the environment is gradually replaced from simulation to experimental prototype to the target.

The goals of the pre-production stage are to verify that all requirements are implemented and the system is working as expected. The system is tested with real hardware and software in a real environment. Examples of test techniques that can be used are real-life testing, random testing and fault injection [13]. Examples of test types in this stage are acceptance tests and safety execution tests [13].

The previous stages have made the system ready for production. The post-production testing ensures that production facilities have high enough quality. The stage is based on development, testing and inspection of production facilities.

2.4 Existing Code

Adding unit tests to existing code that is without tests and not designed for testing can be difficult. For Energy Micro's software there exist only a few tests, which mean that starting to use agile methods like test-driven development may be difficult, and since only a few tests exist there is also a chance that the code is not designed for testability. The ability for testing is important to able to use unit tests and by having unit tests, these can be used as regression testing when software is changed.

2.4.1 Why change software?

There is four reasons for changing software; adding a new feature, fixing an error, improvement of the design and optimize the performance [15].

Adding a new feature is one of the most common reasons to change software, and when adding new features these should always be tested by unit testing. A new feature can be both changing and adding of behavior, which would result in a need to test dependencies.

Testing is important when fixing an error. If there already is a test for this code unit, the test should be changed so it tests the wanted behavior of the function. If not, we should write a test to be able to test the corrections that are done in the code, and to be able to execute regression tests later.

Improvement of the design can result in that features are lost and result in an error. When design is improved without losing behavior it is called refactoring [15]. To be able to do structural changes there is a need to have unit test to perform regression testing. When the structure is changed in the code, there may be need to change existing tests.

Improving the performance of code is like improving design, but the goals is not the same. When algorithm is changed or the structure is changed to optimize the code, it is important to have tests to be able to do regression testing, to ensure that behavior of the code is not changed.

2.4.2 Dependencies

When working with code, dependencies to other libraries and functionality is one of the biggest barriers for testing. When unit testing, you want to use units that are as small as possible, the unit's dependencies can results in many interactions or interaction with functionality that is not available under tests, like hardware. To be able to test a unit you do not want to make any changes to the code, so it is necessary to look for seams in the code. A seam is to change the behavior of the program, without doing any changes in the code you want to test [15]. In C programming there is two good ways to do this.

One of the solutions is to do the modifications of the programs in the preprocessor. The preprocessor can be used to define variables and add addition behavior. It is also possible to use macros to change existing parts of the code. In chapter 3, a macro is used to change the `Delay()` method, so delays do not occur in the loop during testing.

By using static linking it is possible to link other libraries or functions during the linking part of compilation. This can be done with the mock modules that are generated with CMock. Instead of using the actual module, the mock module is linked to create the test binary.

2.4.3 TDD with existing code

Starting to use test-driven development with existing code without tests, can be a problem to find where to start. In “Test-driven development for embedded C” [16], a policy for adapting test-driven development is suggested for a team that has code without tests:

- For new code, use test-driven development.
- Before modifying the code, add tests for the existing code.
- Test the code after modification.

The first point states that when new functionality is developed, the development will follow the default cycle of test-driven development, where the cycle starts with a failing test. The second point suggests that tests for the existing code must be written before any modification is done to the code. When tests have been added, the code can be modified to support new functionality. After modification the tests can be run. After this modification, more of the source code is covered by unit tested and the developers have tested both new and modified code with unit tests. The goal when using TDD on existing code should be to get the module that is changed under test.

CHAPTER 3 EMBEDDED UNIT TEST FRAMEWORK

This chapter discusses possible solutions for testing embedded software. Energy Micros software library consists of different layers, but the most important part is testing of examples projects that are delivered to the development boards.

3.1 Introduction

The purpose of testing the software library is to look at problems with unit testing of a software library for more than hundred microcontrollers, and to find solution to these problems. This chapter will also look at how suitable Unity and CMock is for testing a library like this. If a sufficient part of the software library can be unit tested, it is possible to use these tests for regression testing, since regression testing of the software library is impossible with more than a hundred microcontrollers after a modification in the library. The section will mostly focus on software unit testing, but in some parts also look at software integration testing and integration testing. In addition it will be necessary to see if the examples for each of the kit can be tested with unit tests.

To be able to run tests that supports different microcontrollers, it will be necessary to look at an automatic build system to make testing easier and reduce the number of tests that are run after each modification, Ceedling is a tool that will be looked into here.

3.1.1 Testing of the software library

The layers that will be looked at during testing are the examples, device drivers, board support package and the Energy Micro peripheral library. Testing of the examples and drivers is the most important, because most of logic is placed in these layers.

There will be not testing of the Cortex Abstraction Layer, since this layer should be tested with access to hardware. The CMSIS Peripheral Support Library will not be tested since it mainly consists of definitions of memory registers and bit fields.

3.1.2 Literature

To be able to solve problems that occur during testing, two books have been used to find ideas for how to solve the problems. During testing of software, different problems will be discovered, and it might be possible that the solution of the problem can be found in these books. The two books are listed below:

- ”Test Driven Development for Embedded C” by James W. Greening [16].
- ”Embedded Testing with Unity and CMock” by Mark VanderVoord [21].

3.2 Dependencies

Dependencies to other modules and libraries are one of the barriers to unit testing. During unit testing the tester wants to test the functionality for the given unit, and is not interested in the effects of the dependencies. Testing with all the dependencies will results in slow tests because they test much more functionality than the actual unit, and the functionality will be tested over and over again. Dependency injection is one way to solve this, but we will need a structure of the code for these, and the method is mostly used in object-oriented languages. Stub functions, fakes and mock modules can be used for this during testing, where these is modules is set by static linking or use of macros.

Below follows an example over the dependencies for the blink example. Figure 3.1 shows the dependencies for the simplest example, which is blink. Blink is actually only a template to start developing on the starter kit. The figure shows the dependencies to other modules, but it does not include inline methods that are defined in other header-files, which also are dependencies. If the inline functions were added there would be at least dependencies to the bit-banding peripheral and the device peripheral.

Figure 3.1 shows that the blink example needs the board support package for the LEDs and Trace. In addition it need CMU (Clock Management Unit) for to set up a clock for the delay functionality in the loop. The board support package for the LEDs needs GPIO to set the pins to be able to turn on and off LEDs on the development board and it also needs to enable clocks in the clock management unit. Both CMU and GPIO have dependencies for assert methods. None of these dependencies are needed when doing software unit testing, since they test much more that the actual unit under test. Because of this it should be possible to use stubs, fakes or mocks to remove all of these dependencies. During software integration testing there will be a need to test all these dependencies, since it will be necessary to test the interactions between the modules.

Figure 3.2 shows the dependencies for an example under test that uses mock modules. They are actually not dependencies but a module the capture the calls and compare them with the data and calls that the test expects. The mock module will in addition inject the response to the unit under test. By doing so it is possible to unit test the software without dependencies. When there is a need to do software integration testing, the mock modules will prevent testing the interactions between the software subsystems. When testing the blink example there will be need for three mock modules; the LEDs on the development board, the clock management unit to verify that the clocks is set as expected, and the trace functionality in the board support package.

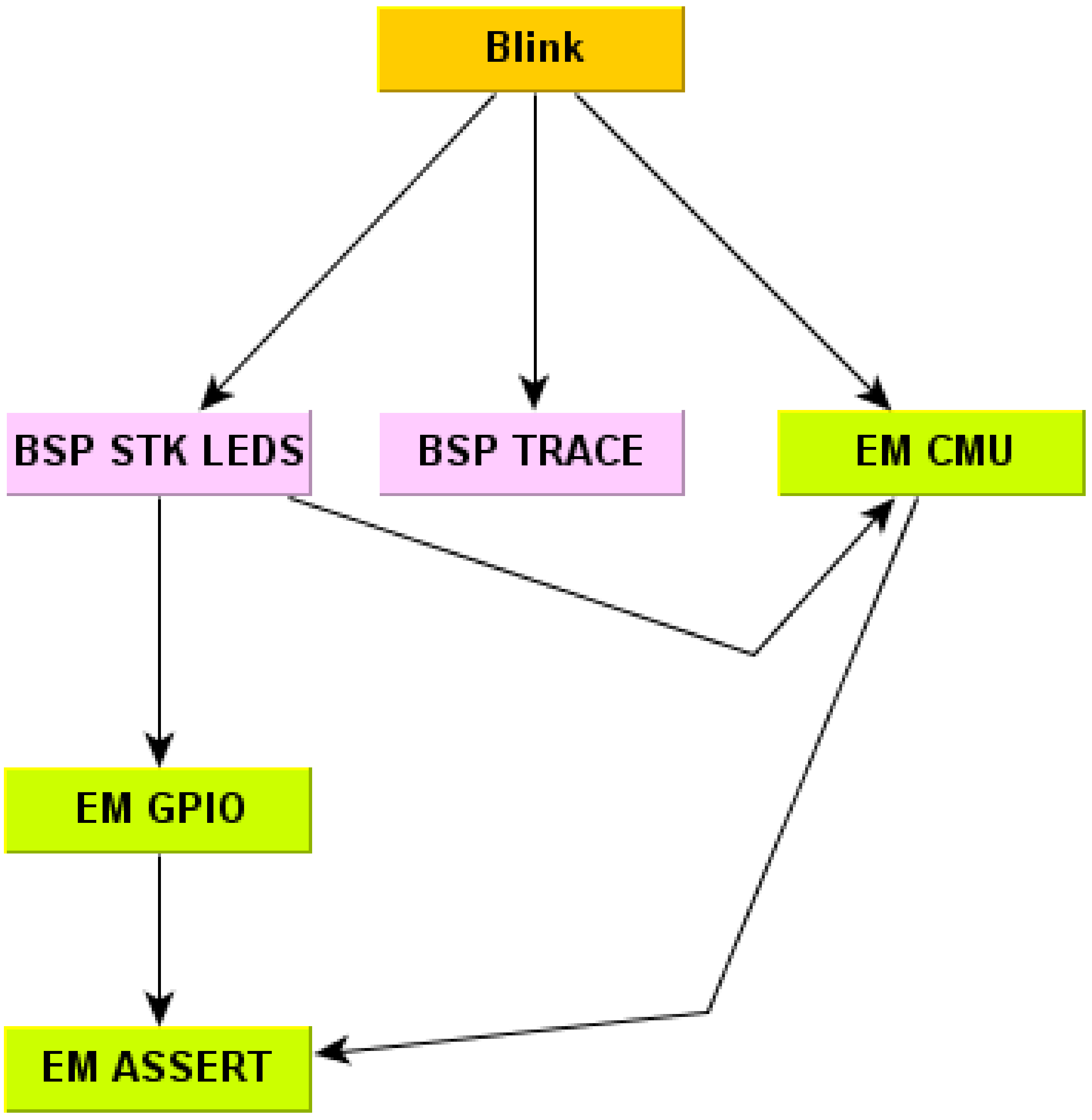


Figure 3.1: Blink Dependecies

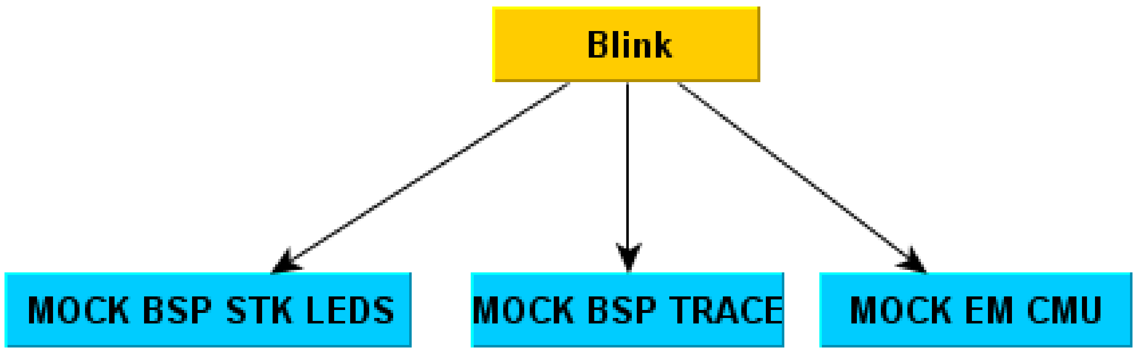


Figure 3.2: Blink Dependecies with Mock Modules

3.3 Software

This section explains the tools, test frameworks, compilers and IDEs that have been used for testing of the software library. Some parts only explains which tools that have been used, and other give a wider description of how these tools can be used.

3.3.1 Unity

This unit test framework is written in the C language [7]. It is a small framework that only consists of a source file and two header files. The framework has features that are special for projects in the embedded space. Unity also includes helper tools. An example of a helper tools is the test runner generator, which generates all necessary code after the tests are written.

Unity has a lot of different assertions to create test for booleans, integer, floats, bitwise, integer ranges, strings, pointers and memory assertions. Integer variables can be tested at given bit-size, hex-values and of signed/unsigned type. Examples of some of these are given below:

TEST_ASSERT_TRUE Assert that a variable is true.

TEST_FAIL Will always return a test failure.

TEST_ASSERT_EQUAL_INT64 Compares two 64-bit signed integers.

TEST_ASSERT_INT_WITHIN Checks that an integer is in a given range.

TEST_ASSERT_BITS_HIGH Checks that the correct bitmask is set to high.

TEST_ASSERT_FLOAT_WITHIN Compares two floating values, with a given fault tolerance.

TEST_ASSERT_EQUAL_STRING Compare that two strings are equal.

TEST_ASSERT_EQUAL_PTR Checks that two pointers are the same.

TEST_ASSERT_EQUAL_MEMORY Checks two blocks of memory, to compare packed structs and buffers.

There are extensions to Unity, which make it possible to use test fixtures. Other assertions and options can be found in the Unity summary [8].

3.3.2 CMock

CMock is a software library used to generate mock modules from C header-files. CMock uses Ruby to generate mock modules from the function declarations that are specified in the header-files [3].

When CMock has generated mock modules, the next step will be to compile and link the mock modules instead of the real modules. Header-files are still used for declaration of the methods. In the tests, the expectations of methods with parameters are set and return values set for non-void methods. The method under test is then called and the unit test is ready for compilation and testing.

When the tests has run it will give feedback if the function was called as many times as expected, if not it will return that it was called less or more times than expected. Return values will be injected when this is specified. All parameters will be asserted against the expectations that were set. It is also possible to configure CMock to verify that methods are called in the correct order.

Parsing

CMock is limited in parsing C. The way CMock do parsing is to use regular expression to parse the header-files. This limits the support to what it actually needs, but anyway it supports most of the necessary functionality. One point where CMock lacks support is parsing of custom data types, for examples structures, which becomes difficult to compare. Instead of parsing header-files with regular expressions, CMock could have used a C parser written in ruby, but currently there is none that supports most of C99 language. The developers have earlier announced that support for structure is on the way [9].

Mock types

CMock has five mock module types that can be generated. These are expect, array, callback, Cexception and ignore [4]. Expect is the basic function, that is used in most of the tests, it defines that the function shall be called, and all the parameters to the method are asserted. The array type takes an array as parameter, together with the size of the array and asserts all of these elements. The callback function will take a callback function as a parameter, and call this stub function when the mock is called. Cexception is a function that can be used together with Unity and CMock to create exceptions, Cexception is not used in this thesis. The last function is ignore, which ignore all calls to the mock function. An example of each of these functions is showed in Listing 3.1.

```c
/* Expect */
void print_Expect(params);
void print_ExpectAndReturn(params, return_val);

/* Aray */
void print_ExpectWithArray(*ptr params, size);
void print_ExpectWithArrayAndReturn(*ptr params, size, return_val);

/* Callback */
void print_StubWithCallback(CMOCK_print_CALLBACK callback);

/* CException */
void print_ExpectAndThrow(params, throw_value);

/* Ignore */
void print_Ignore();
void print_IgnoreAndReturn(return_val);
```

Listing 3.1: Mock Module Types

Configuration

The configuration of CMock is done with YAML , where the configuration file is used as input to CMock. Some of the features that can be configured is that attributes can be ignored, configuration of which plugins to use for mock types, how types and custom types shall be threated and configuration of how strict the mock module should be with verifying call in the correct order. For a full overview of configurations is CMock, see the documentation [4].

Structures

The biggest problem with CMock is how it handles comparisons of structures. CMock do comparisons of structures, by comparing the memory byte by byte. This will result in problems when the structures are not packed. On embedded systems, structures will often be used as a memory register, and packing memory register can result in incorrect testing since packing a structure will result in a register that is different from on the embedded system. This can result in testing on a model that does not represent the actual model on the hardware. An unpacked structure will contain garbage values where data alignment/padding is necessary, which will make a test fail. There are some ways this can be solved:

- If it is possible to pack the structure, add *-fpack-struct* as an option to GCC, which will force that the structure is packed.
- Create a helper method for Unity that do the assertions, and add this to the configuration file for CMock.
- One possible way is to initialize the structure with `calloc(size_t, size_t)`, which will initialize the memory block as zero.
- If it is not important to test the feature, ignore the mock module.

Inline functions

Inline functions are often used when programming in the C language, for functions that are often used. Since function call are expensive, inline functions can be used instead so the function are included in the code, instead of using a function call. `static inline` can be used to be make the definition private to one translation unit[1].

In Energy Micro's software library `static inline` is often used, which will make the code faster og by that use less energy. When generating mock modules with CMock this is a problem.

CMock ignores `inline` functions by default, because if a mock function of the function was created, it is not defined which of the functions that will be used by the compiler [10]. The way to solve this is to wrap the definition into a macro and define a function that will be used during test, as shown in Listing 3.2.

[1]A Translation unit is the output from preprocessing, that will be used by a compiler to create an object file.

```
#ifdef TEST
void func_write(uint32_t *addr, uint32_t value);
#else
__STATIC_INLINE void func_write(uint32_t *addr, uint32_t value)
{
  *((volatile uint32_t *)addr) = (uint32_t)value;
}
#endif
```

Listing 3.2: Redefinition of Inline Method when Using Mock Modules

3.3.3 Ceedling

Ceedling is an extension to RAKE [6], so it works as a build system for C projects. Ceedling is created to support test-driven development in C and is designed to connect Unity and CMock [2]. The functionality that Ceedling can support when running tests, is to list all test tasks, run all tests, run a specified module or run tests that depends on changes since last time the tests was run.

3.3.4 Compilers & IDEs

In addition to software used for testing there have been used several IDEs and compilers. GCC have been used for compiling and linking the tests that are run on the host computer. Compiling and upload in Windows was done with IAR, in Linux have CodeSourcery been used to compile the examples.

The ruby language has been used to create necessary script and to run the scripts that are used by Unity, Ceedling and CMock.

3.4 Testing of Peripherals

This section covers testing of a few peripheral APIs. The section will look at features that are difficult or impossible to test, and how to solve some of these problems. The APIs that was tested was the watchdog, the peripheral reflex system and the AES (Advanced Encryption Standard) peripheral. All the source code in this chapter is taken from the Energy Micro's software library, and modified to enable testing of the functionality.

The peripheral APIs contains little logic, and is mainly a hardware abstraction layer. The value of testing this functionality outside the hardware has little value. But it will still be necessary to test this on a local computer to support test-driven development, where testing is performed during development.

3.4.1 Watchdog

The watchdog is a peripheral that will reset the system in case of a system error. The main functionality for the watchdog is that it uses a clock to count towards a timeout, without resetting the timer, the watchdog will trigger a timeout. The watchdog can be initialized with different settings for energy modes, extern clocks, enabling, locking and period for timeout.

Unit testing

Most of the assertions that were done by the unit tests in to check that correct bits are set, since the main functionality for the watchdog code makes changes to the control register for the watchdog. Reset of the counter is done by writing a bit to the command register.

The watchdog do write operations to the low frequency domain, and need to wait to all previous write operations are finished. Waiting is done by a while-loop that checks if the $SYNCBUSY$ register is set. There is no good solution to test this, one possible solution is to test this is to use threads. One thread will be used to go into the waiting-loop, while the other thread is used to clear the $SYNCBUSY$ register and verify that other thread waits until the register is cleared. Another solution is to change the code structure to make it more testable, which can be done by adding methods that read and write to the $SYNCBUSY$ register, by having a method for accessing the register, it is possible to create a mock object, where the return values can be injected. In this way several expectations can be set and verified. The best solution to check that this is done is to test the feature on hardware and by using code review, but unit test to be run for regression testing may also be important.

A mock module was generated with CMock and used for the bit-banding, because the mapping between the peripheral registers and the bit-banding is not available without hardware. Use of a mock module work, but the generation had some problems, which is described below.

Problems with testing

In context of the whole software library, the watchdog is a small and simple peripheral device. During unit testing of the software functionality for the watchdog, we found a few problems. Which are listed below.

Bitbanding Bit-banding is a technique that is used to modify single bits in memory registers or memory. With bit-banding it is possible to write a word to a memory location, the hardware is then responsible for setting the correct bit, by using this technique updating a bit can be done in one operation.

The bit-banding module consists of two static inline functions that take the address of a register, the bit in the register and the value as parameters. The function will then calculate the memory address in the bit band and write the value as a 32-bit unsigned integer. The implementation can be seen in Listing 3.3.

```
__STATIC_INLINE void BITBAND_Peripheral(volatile uint32_t *addr, uint32_t
    bit, uint32_t val)
{
  uint32_t tmp = BITBAND_PER_BASE + (((uint32_t)addr - PER_MEM_BASE) * 32)
      + (bit * 4);

  *((volatile uint32_t *)tmp) = (uint32_t)val;

}
```

Listing 3.3: Implementation of BITBAND_Peripheral

Since writing from the bit band is a hardware feature, this will not happen during unit testing on a computer. To verify that the software actually modifies the bit, there are three possible solutions:

- Create a mock module/object that sets the expected behavior, to check that functions were called with the correct parameters.
- Allocate memory for the bit band, and use unit test assertions to check that the values are set in the memory address.
- Create a function to be used under testing, that takes the same parameters and then set the correct bit in the watchdog memory registers, and run the unit test assertions. An example is shown in Listing 3.4. Creating a function for this can be dangerous since you also add functionality that only are used during testing, which means that errors could occur in the test function and we get more code to maintain. It is also possible that the implemented code is not correct according to how the hardware works.

```
#ifdef TEST
void BITBAND_Peripheral(volatile uint32_t *addr, uint32_t bit, uint32_t val
    )
{
  uint32_t tmp = (uint32_t)*addr;
  if (val)
    tmp |= (1 << bit);
  else
    tmp &= ~(1 << bit);

  *addr = tmp;
}
#else
__STATIC_INLINE void BITBAND_Peripheral(volatile uint32_t *addr,
                                        uint32_t bit,
                                        uint32_t val)
{
  uint32_t tmp = BITBAND_PER_BASE + (((uint32_t)addr - PER_MEM_BASE) * 32)
      + (bit * 4);

  *((volatile uint32_t *)tmp) = (uint32_t)val;
}
#endif
```

Listing 3.4: Redefinition of BITBAND_Peripheral

Mock module for bit-band Automatic generation of mock module with CMock for the bit-band header-file is not possible without modification of the header-file before generation. CMock takes header-files with function prototypes as input.

The bit-band functions are defined as `static inline` to improve the performance for this functions, since function calls are expensive. The compiler will try to inline the function in the code when `inline` is used. In the C language, `static inline`-functions are often implemented in the header-file, so it is possible to include them.

CMock do not accept implemented functions as input, which is a problem. Another problem is that CMock also ignore functions defined as `static inline`. It is therefore necessary to modify the header-file before generation to only be function-prototypes, and replace the `static inline` with `void`. After this is done it is possible to automatic generate mock modules.

When the input file for generation is modified, it is necessary to do other modifications, to actually use the mock module. The function prototypes used must be included in the header-file for the mock module. The functions that actually use bit-band, must include the header-file for the mock module, instead of the original file, therefore the include must be modified, an example of this is shown in Listing 3.5.

```
#ifdef UNITTEST
#include "Mockem_bitband.h"
#else
#include "em_bitband.h"
#endif
```

Listing 3.5: Macros Used to Set Mock Module Under Test

Syncronation to low frequency domain A problem is that *SYNCBUSY* is defined as read-only (`violatile const`), this can be modified by wrapping the definition into a macro where it will not be defined as read-only. Another solution is to do an implicit cast to `void *`, which is not a safe method. This will overwrite the data and may result in the application crash, depending on where the data is stored. Listing 3.6 shows how the access restrictions are changed during test, to make it possible to write to values that are defined as read-only.

```
#ifdef __cplusplus
  #define    __I     volatile             /*!< Defines 'read only'
     permissions                  */
#elif TEST
  #define    __I     volatile             /*!< Set read/write under testing(
     C only)             */
#else
  #define    __I     volatile const       /*!< Defines 'read only'
     permissions                  */
#endif
#define      __O     volatile             /*!< Defines 'write only'
   permissions                  */
#define      __IO    volatile             /*!< Defines 'read / write'
   permissions              */
```

Listing 3.6: Modification of Access Restrictions

Memory registers The memory registers for the watchdog is given the address it has on the bus on the microcontroller. To be able to run test, it is necessary to give it a valid address on the computer that it is executed on. This is done by declaring a pointer, see example in Listing 3.7. For the `setup(void)` for the test it is now possible to allocate memory to the memory registers, which can be done by using `calloc(size_t,size_t)`. This will initialize the memory to zero. When using `calloc(size_t,size_t)` it will be necessary to call `free(void*)` deallocate the memory block, this is important to avoid memory leakage.

```
#ifdef TEST
WDOG_TypeDef *WDOG;
#else
#define WDOG          ((WDOG_TypeDef *) WDOG_BASE)          /**< WDOG base
   pointer */
#endif
```

Listing 3.7: Modify Address of Memory Register

Hardware specific functionality All watchdog functionality cannot be tested, since it depends on hardware. The reset signal that is given when the watchdog times out is not possible to test, since it only triggers an interrupt which results in a reset. This is functionality that is done by the hardware, and cannot be tested by using unit tests.

One of the registers in the watchdog is `SYNCBUSY`, this is a register where data is set by the hardware during writing to the low frequency domain. When hardware is not available, this functionality cannot be tested without modification.

The mapping between memory registers and the bit band for peripherals is hardware dependent, so this mapping will not work under testing.

3.4.2 Peripheral reflex system

PRS (Peripheral Reflex System) is a peripheral API that makes it possible for peripheral devices to communicate without use of the CPU. The PRS has 12 channels that can be configured for sending pulse or level signals [11]. The API consists of methods that are used set signal sources, level and pulse triggers.

The first part that was necessary to fix before testing, was to define a pointer for the type declaration for the PRS registers, instead of using a base pointer that would result in segmentation faults. This was done in the same way as in Listing 3.7. The next step is to create a test file with set up and tear down, that first initialize the PRS registers using `calloc(size_t, size_t)` to have zero-initialized register, in the tear down it is necessary to free this memory block with `free(*ptr)`. The next step was to create all the tests and generate a test runner and then compile and run tests.

Since the methods in the API only writes to registers that are read- and writeable, testing of this peripheral was easy. Methods are called with given parameters, and assertions are done on the registers afterwards to verify that they were set correctly. Testing the channel arrays in the registers worked also without any problems. The purpose of having unit tests for this API is to be able to do testing during development, to verify that no functionality was broken.

3.4.3 Advanced encryption standard

AES (Advanced Encryption Standard) is an encryption standard that is implemented in Energy Micro's microcontrollers. AES is added as en peripheral device to reduce the need of CPU usage during encryption and decryption and thereby reduce energy usage. The microcontroller support 128-bit and 256-bit encryptions [11]. The API consists of methods for configuration of interrupts, and the methods that are necessary for encryption and decryption.

Under testing of the AES peripheral API two problems was encountered. The first problem was that the tests would not compiled since a reverse instruction was used, that was not available in the x86 instructions set, which is used when running tests on the host computer. The second problem is that data is read and written to registers that are not memory mapped. When data is written here, the hardware does the shifting of the data internally, in this case barrel shifting. To be able to test that the correct data is read or written, hardware is necessary, or it must be possible to compare values in some way.

Instructions

The *rev* instruction that results in compilation errors is an instruction that is used to reverse the byte order of a word in the ARM instructions set. Since these tests are run on the x86 processor and compiled with GCC this instruction will not exist. Since the behavior of the instruction is important for the behavior of the unit, it must be replaced with code that does the same operation. Listing 3.8 shows how this is fixed, where the method is chosen depending on if *TEST* is set in the preprocessor or not. It should also be possible to solve this problem using a macro instead of having to wrap the CMSIS method for test instrumentation. Adding code for test purpose can be a pitfall, because errors can occur in this code too. Using a mock module to manually inject the return value is also possible. When mock module is created for one function in CMSIS, the function will also need to be moved out of the header-files, which may result in code that is difficult to understand and maintain. A possible solution is to create a library of stub functions for testing, which can be used as input to CMock.

```
#ifdef TEST
__STATIC_INLINE uint32_t __REV(uint32_t value)
{
  return ((value>>24)&0xff) |
  ((value<<8)&0xff0000) |
  ((value>>8)&0xff00) |
  ((value<<24)&0xff000000);
}
#else
__attribute__( ( always_inline ) ) __STATIC_INLINE uint32_t __REV(uint32_t
    value)
{
  uint32_t result;

  __ASM volatile ("rev %0, %1" : "=r" (result) : "r" (value) );
  return(result);
}
#endif
```

Listing 3.8: Reverse Instruction for Test

Data registers

The methods in AES API write directly to a structure member that is not memory mapped. The data-register is 128-bit, but only 32-bit is accessible in the structure, and the hardware is responsible for the barrel shifting. By the current structure of the code it is impossible to test that the correct values is passed to the register under test, only the last 32-bit can be asserted. A possible way to solve this is to use set/get or read/write methods to access the registers. With these methods it is possible to generate mock modules that can be used to verify that all assignments to the data register is correct. By using inline methods, the only difference should be that there are a few extra methods and that the performance only is different during compilation.

Listing 3.9 shows an example of how methods can be used to access data registers in a structure. The structure `AES_TypeDef` includes a member `DATA`, which is a register on an embedded device. When data is written to this register, the hardware will do the processing of the data. Reading data with `read_write(uint32_t)` during the unit tests will only be able to assert the last value that was set. By adding read and write methods to the `DATA` member with the methods `AES_DataRead(void)` and `DataWrite(uint32_t)`, it is possible to create mock modules that can be used to verify expected calls and inject the return values that are necessary for further processing in a method.

```c
typedef struct
{
  __IO uint32_t DATA;
} AES_TypeDef;

__STATIC_INLINE uint32_t AES_DataRead(void)
{
  return AES->DATA;
}

__STATIC_INLINE void AES_DataWrite(uint32_t data)
{
  AES->DATA = __REV(data);
}

void read_write(uint32_t *in)
{
  int i;
  uint32_t *out;

  for (i = 3; i >= 0; i--)
  {
    AES_DataWrite(_iv[i]);
  }

  for (i = 3; i >= 0; i--)
  {
    out[i] = __REV(AES_DataRead());
  }
}
```

Listing 3.9: Access to Data Registers

3.5 Use of Mock Modules

Mock modules/objects are essential when testing is performed without access to hardware. To be able to have automated testing of all the binaries that are created for different IDEs, examples and development kits, it is necessary to use these mock modules.

Mock modules will be used to verify that calls are done to the hardware, and it is also used to inject values that are to be returned on specified function calls, where the function returns data. By using these mock modules, the hardware will not be needed, since no values are written to memory registers on the hardware, but are stored into the mock module for verification.

It is possible to manually write mock modules, but since emlib consist of many hundred functions, this will take too long time, and it will be too difficult to manage modification of the library over time. CMock will be used to do the generation automatic, which is done in under a minute. Another advantage is that this can be run each time, so the mock modules are updated with new functionality that may have been added.

3.5.1 Energy Micro peripheral library

Energy Micro Peripheral Library (emlib) is the HAL (Hardware Abstraction Layer) in the software library. The HAL's task is to be a programming interface and support all the peripheral devices. Since emlib do all the calls to the hardware, mock modules can be generated for emlib and be used instead of using the emlib. With the mock modules, the example project can be tested automatically and without use of hardware. In this way developers will not be able to test the functionality in the modules that were mocked, but the functionality that uses the mock modules can be tested without dependencies.

After an attempt to use the mock modules it was discovered that mock modules for some parts of the Cortex abstraction layer and some of the peripheral support library also was necessary. Later it was also discovered that we needed to create mock modules for board support package and the driver to remove the dependencies in the examples. The biggest problem with the dependencies in the examples was when using the segment LCD driver. Only for initialization, the segment LCD has over twenty function calls to LCD API, and writing to the text segments will results in even more calls to the LCD peripheral API. This is unnecessary and will result in too much time used on write tests for these functions.

3.5.2 Generation of mock modules

Since many of the functions that exist in emlib are `inline_static`, it is impossible to generate mock modules from emlib without modification, since these functions must be included in the mock modules. To solve this, a Ruby script was written. The main task for this script is to wrap all inline static functions into preprocessor macros that use void functions instead. The header-file is then preprocessed with the GCC preprocessor, and after this the header-file can be used to generate mock objects.

Include directives for header-files is important when the file is used in compilation, but during preprocessing to generate mock objects, these includes will only generate conflicts and functions that are not needed for generation of mock modules. Therefore it is also necessary to wrap the include directives into #if directives so that they not are included during preprocessing of mock objects. But some of the include files are necessary since they include definitions that are used to define methods. These includes are accepted by the script.

The script is run by specifying a YAML configuration as input. This configuration file specifies paths to all necessary include files, paths where the files shall be saved after generation and paths to CMock. The source can be found in appendix C. The script only works on the emlib, and some manual work is still necessary to be able to generate mock modules. The script was created to automate some of the process during generation of mock modules.

The output from running the script will be the header and source file for the mock modules, which will be used instead of the emlib. The script will also generate new header files that must be used instead of the header-files in the emlib. This is important because `inline_static` functions will create conflicts during compilation, so these header files must be used.

Generation of mock modules for other parts than the peripheral library (emlib) was done manually using CMock. Scripting of all parts will be necessary during automatic building and testing. The reason for this is that changes can have been done to the library, and this will require new mock modules. We should therefore generate new mock modules each time all binaries and tests are compiled.

3.5.3 Testing with mock modules example

Listing 3.10 and Listing 3.11 shows some example code and tests using mock module. The behavior of `click_left(void)` is that a variable is incremented, an LED is toggle and a text containing the number of clicks is written to the segment display. The test function for `click_left(void)` will first set an expectations for that a LED is toggled, the first parameter tells which LED to toggle, and then the return value for method, since the method returns an integer for a status code. The next line sets the expectation for that `SegmentLCD_Write(char*)` is called with the expected parameter "1 click". The next line is the function call, and afterwards the mock modules are verified, to check that all the expected functions are called with the correct parameters. The test runner which is generated will have the methods for running each of these methods and do the initialization and verifying of the mock modules. The last assertion is a unit test that verifies that the left click variable has been incremented.

```c
static uint32_t leftClick = 0;
char output[8];

void click_left(void) {
  leftClick++;
  BSP_LedToggle(1);
  snprintf(output, 8, "%lu CLICK", leftClick);
  SegmentLCD_Write(output);
}
```

Listing 3.10: Code Under Test with Mock Modules

```c
void setUp(void)
{
  leftClick = 0;
}

void tearDown(void)
{
}

void test_click_left(void) {
  BSP_LedToggle_ExpectAndReturn(1, 0);
  SegmentLCD_Write_Expect("1 CLICK");

  click_left();

  TEST_ASSERT_EQUAL_UINT32(1, leftClick);
}
```

Listing 3.11: Test Using Mock Modules

3.6 Testing of Examples

This section will look at possible ways to perform testing of the examples that are created for the starter and development kits. Testing will be performed in two different ways:

Mock modules By using this method, tests will be run on the host computer, and the tests will not have access to hardware. Instead of using emlib, the test will use mock modules for emlib. These mock modules are generated by using CMock.

Development board The tests will be run on the development board, which means that the software have access to the hardware, and the unit tests will be able to compare values in the memory registers. While running tests on the development board, the test results will be written back to the host computer using SWO (Serial Wire Output).

Both of the methods will have advantages and disadvantages, which will be covered later in this section. The blink example in Listing 3.12 is taken from Energy Micro's software library.

3.6.1 STK3700

All the examples that are tested are created for the EFM32GG STK3700 starter kit. Currently there exist 24 different examples for this starter kit, and most of them have support for all the IDEs that was mentioned in the background chapter. Compilation of these examples it will result in nearly a hundred different binaries that need to be tested. Figure 3.3 shows an overview of the STK3700 starter kit, with some of the features that are available on the board, the figure is taken from the STK3700 user manual.

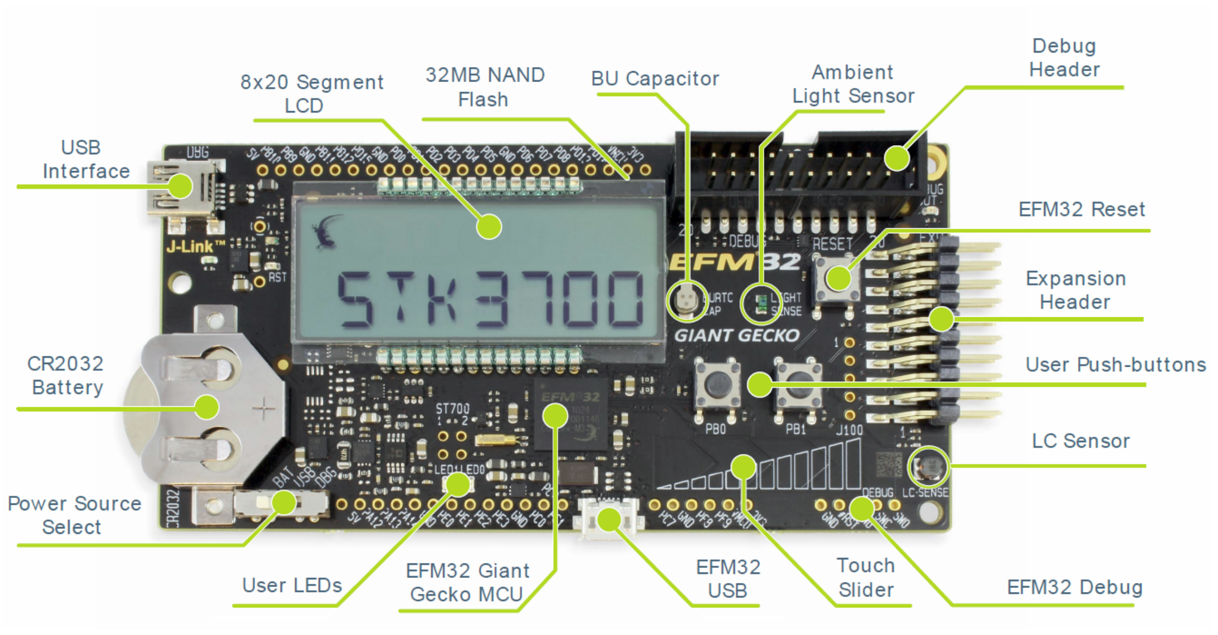


Figure 3.3: STK3700 Hardware Layout [12]

The starter kit is based on the Giant Gecko microcontroller, and is suitable to test many of the features that this microcontroller supports. The example tests will use some of the examples that are shipped with the microcontroller.

3.6.2 Testing with mock modules

When mock modules are generated instead of using the CMSIS and emlib, it will no longer be possible to detect errors in these layers during testing. But the mock modules will be able to verify that the correct function calls are run in the examples. The advantage of testing without the use of hardware is to make testing automated, which will save time. Testing of emlib and CMSIS can be done by other tests, so we can presume that these layers are without errors, and focus on the other layers that are used by the examples. By testing with mock most of the dependency is removed during software unit testing. Software integration testing is needed when the software is tested layer by layer. Below three of the examples are tested using mock modules.

Testing of blink example

The first test of the example projects uses the blink example, which is a simple template to start developing for the starter kit. The only additional functionality in addition to initialization is the LEDs that are blinking.

Listing 3.12 shows the source code of the example, the first issue that needs to be fixed before testing is to rename the main function, since the entry point is needed for the tests. This is solved by defining the main function as `app_main(void)` if *TEST* is defined. The next changes that was done was to move the infinity loop to a new method, that was named `blink_loop` and call this function from the main method. To prevent the loop from never stopping, the while loop was change to a do-while loop, to ensure that it run at least run once. Above blink_loop(void) some test instrumentation is added. The first thing is to define INFINITY to zero during testing and one else, the last define is a macro that set the delay to zero under test. Setting delay to zero is necessary during test, since there is no hardware that can create interrupts and increment the variable during a system tick. The source code can now be tested with unit tests and mock modules.

```c
volatile uint32_t msTicks;

void SysTick_Handler(void)
{
  msTicks++;
}

void Delay(uint32_t dlyTicks)
{
  uint32_t curTicks;

  curTicks = msTicks;
  while ((msTicks - curTicks) < dlyTicks) ;
}

#ifdef TEST
#define Delay(x) Delay(0)
#define INFINITY 0
#else
#define INFINTY 1
#endif

void blink_loop(void)
{
  do
  {
    BSP_LedToggle(0);
    BSP_LedToggle(1);
    Delay(1000);
  }  while (INFINITY);
}

#ifdef TEST
void app_main(void)
#else
int main(void)
#endif
{
  CHIP_Init();

  BSP_TraceProfilerSetup();

  if (SysTick_Config(CMU_ClockFreqGet(cmuClock_CORE) / 1000)) while (1) ;

  BSP_LedsInit();
  BSP_LedSet(0);

  blink_loop();
}
```

Listing 3.12: Blink Source Code

Listing 3.13 shows the tests for the blink example. The file shows three different tests where the first function is testing of the main method, where also the loop is run once. Most of the code lines in the test are expectations for which functions that shall be called by the main method. When all expectations and return values are set, the main method is run. All verifying of the mock modules is done by the test runner, which is not visible here. The second test verifies the blink loop and the expectations for these methods are that both LEDs are toggled. The last test asserted that `msTicks` is incremented by one, when an interrupt occur.

```c
void test_app_main(void)
{
  CHIP_Init_Expect();
  BSP_TraceProfilerSetup_ExpectAndReturn(0);
  SysTick_Config_ExpectAndReturn(0, 0);
  CMU_ClockFreqGet_ExpectAndReturn(0x040020,10);
  BSP_LedsInit_ExpectAndReturn(0);
  BSP_LedSet_ExpectAndReturn(0,0);

  BSP_LedToggle_ExpectAndReturn(0,0);
  BSP_LedToggle_ExpectAndReturn(1,0);

  app_main();
}

void test_loop(void)
{
  BSP_LedToggle_ExpectAndReturn(0,0);
  BSP_LedToggle_ExpectAndReturn(1,0);
  blink_loop();
}

void test_systick(void)
{
  uint32_t temp = msTicks;
  SysTick_Handler();
  TEST_ASSERT_EQUAL_UINT32(temp + 1, msTicks);
}
```

Listing 3.13: Blink Test Code

The changes and tests above make it possible to do unit testing of the example, where almost all code was tested. The only method that not was tested was the delay function, which is impossible to test without hardware or use of threads.

Testing of clock example

The clock example is an application that uses the segment LCD to display the time, buttons to set the time, and a real time counter that updates the clock each minute. For each iteration the clock enters an energy efficient mode, which pause the loop, and the system wakes up by either user input from the button or the real time clock. Interrupts and entering different energy mode do not happen during test, since hardware is not available, so calls are verified by mock modules.

Some changes need to be done to the source to make it possible to run the tests. The main method must be renamed and the while loop must be changed to a do-while that do not run infinitely during test. This example already has a loop method, so this is not needed here. To be able to run methods and do assertions from the test file, a header-file is needed, this file is shown in Listing 3.14. In the header-file, variables are defined as `extern` and methods are declared so the unit tests can access these.

```c
#ifndef CLOCK_H
#define CLOCK_H

#include <stdbool.h>

extern uint32_t hours;
extern uint32_t minutes;
extern bool oldBoost;

void GPIO_ODD_IRQHandler(void);
void GPIO_EVEN_IRQHandler(void);
void RTC_IRQHandler(void);
void clockLoop(void);

#ifdef TEST
void app_main(void);
#else
int main(void);
#endif

#endif /* CLOCK_H */
```

Listing 3.14: Header-file for Example Test

The mock modules are used to verify that correct calls are done, and to inject necessary return values. Mock modules are used for the peripheral APIs and the *VDDcheck driver*. The test covers all of the code in this example, no dependencies in other drivers and peripheral APIs are tested since mock modules are used instead. The clock example has also some logic that is tested by unit tests. This include that time is set correct. Being able to test all units and methods in the example should verify that the example works as intended in the tests. That it works together with other software components and hardware must be tested with software integration tests and integration testing.

Testing of NAND flash example

The last example that was tested using mock modules was the NAND flash example, which is used to perform commands for reading and writing to the NAND flash memory on the microcontroller. The commands are given to the development board and microcontroller by writing commands over a serial interface. The examples demonstrate features in the NAND flash driver. The example was difficult to test because of many definitions that need to be changed for testing.

To be able to run this example on the host computer it was necessary to generate mock modules for the NAND flash driver and the driver for the serial interface. It uses some of the mock modules of API in emlib. In addition we could have created mock modules for features that are used in the standard libraries of C. If verification of parameters to `printf(...)` and `putchar(char)` is not important, stub functions can be use instead.

Like in the other examples there was need to change some of the structure, to be able to do the testing. Function declarations were moved to a header-file, test instrumentation to rename main methods was added and the loop was moved out of the main method. With these changes it became possible to compile and start writing tests for the system.

In this example we used calls to methods in the standard libraries of C. `printf(...)`, `putchar(char)` and `getchar(void)` are used to write and read text over the serial interface. The functions used for text output can easily be ignored using a macro to remove output to the test output. Example of this is shown in Listing 3.15. But `getchar(void)` is needed to read input, which is used by the loop to execute the command. Since the commands need to vary during the test, this cannot be fixed by a macro. There are two solutions to this problem, either create a mock module, which can be used to inject values to `getchar(void)` or ignore the `getCommand(void)` which is used to read commands and put them in a command queue, make the command queue available in the tests and manipulate them before test. This way it is possible to test all the functionality expect for `getCommand(void)`, which can be tested if mock modules are created. The functions `printf(...)` and `putchar(char)` can also be mocked to verify that the correct values are printed.

```
#define printf(...) printf("")
#define putchar(x) putchar('\0')
```

Listing 3.15: Redefinition of Functions with Macros

Segmentation fault was found during testing of the example. The problem is the use of the DWT (Data Watch point and Trace) unit. The first issue is that DWT is turned on in the core debug register in the CPU. This register is defined as a pointer to the register. To be able to test it, the register need to be defined as `CoreDebug_Type` pointer and memory allocated for the definition. The definition is changed in CMSIS and memory is allocated in the setup method in the tests. In addition we need to change two variables that are defined as pointers to a memory address; `DWT_CTRL` and `DWT_CYCCNT`. These were changed to 32-bits unsigned integers during test and the variables was added as `extern` in the header-file to be accessible for assertions by the tests.

It was possible to cover most of the code with tests using mock modules and unit tests. Mock modules were used for expectations and return value injection for all method calls to the drivers and the peripheral API. Unit test was used to assert that values were set correctly by the code. Most of the variables were not visible outside the source file, so `extern` was added for these values in the header-file during test, for `static` variables, the `static` keyword was removed to make it accessible for other source files. All changes to variables was put into preprocessor directives that checked if *TEST* was defined.

3.6.3 Testing on development board

By running the unit tests on the development board, it will be possible to find errors in all levels of the software library, and it might be possible to detect errors in the hardware too. IAR where used as an IDE during testing on the development board. By testing the example, all the layers are used and software integration testing is done, in some degree integration testing too, since hardware is used. To be able to see the results of testing, SWO (Serial Wire Output) was used to write the test results back to the IDE.

To be able to do assertions on the development board, Unity was used. To be able to run the tests, some modifications were needed in the example code. The main method was renamed, and the loop was moved out of the main method and changed to a do-while loop, so the loop would not run infinitely. Unity compiled with IAR, and run as expected on the hardware. Assertions were done on registers that were readable, to verify that the example had modified the registers as required.

Mock modules were tested on the development board. The mock module for the segment LCD was tested to verify that it worked in the same way as when testing on a host computer. The segment LCD was tested and worked as expected. Using mock objects on the development board may be necessary to test some functionality and to make it possible to automate some testing, without having to depend on user interactions or additional hardware.

Automation and scripting of testing several examples on hardware might be possible, and it is possible to use other test environments than an IDE and SWO to get test results back to the test host. This project did not look into how to automate this. Testing on the development board was done to see if it was possible to run automated unit tests on the development board, and to verify that the test frameworks Unity and CMock worked as expected on the board.

The advantage of running the test on the development board is that the code runs on the intended environment with hardware. When hardware is in place, most functionality can be tested, and there is no need to remove functionality like the delay method, since the interrupts is not working on the host computer during unit testing. But it could be necessary to change the delay during unit testing, since a second delay in a unit test is bad practice, because the tester will have to wait longer for the tests to finish. The disadvantage is that there still is not any user interaction during automated testing, and that not all register can be read and asserted.

One important issue when using assertions on hardware is that not all registers can be read, since they are only writeable. This means that other actions must be taken do test the feature, since a write operation to a register will result in some behavior by the hardware. It will work to verify that the effects are as expected, which can be asserted either manually or automatic, which means that it can be asserted by a test harness that is reading other registers or by the developer. Cases where an operation is done and the effects of the operation done by the hardware is verified, are integration testing, where both the software and hardware is tested.

Some registers are also mapped to other register using bit operations. When a *LED* is toggled, a bit is written to the register *DOUTTGL*, and the register output can be read from *DOUT*. Example is shown in Listing 3.16.

```
GPIO->P[4].DOUTTGL = 1 << 3;

TEST_ASSERT_EQUAL_UINT32(8, GPIO->P[4].DOUT);
```

Listing 3.16: Assertion on Development Board

3.7 Testing of Drivers

The board support package is a driver to support the functionality of the board controller and the devices on the development board. The drivers are collections of methods that are often used by the examples to use functionality in emlib. Two drivers was tested, the first was VDD voltage check, which is used to check the voltage to the microcontroller. The other was the LEDs driver for the starter kit.

3.7.1 VDD voltage check

This driver is used to check if the voltage in the microcontroller is below a given value. This is used for example in the clock example to check if it is necessary to boost the voltage to the segment LCD.

It is straight forward to write tests for this driver. Unit tests can be used to assert the return values from the function, and expectations is set for the function calls to the voltage comparator mock module. This is done for each of the methods in the driver.

The function `VCMP_Init(VCMP_Init_TypeDef)` takes a structure as parameter. When a mock module is created for this, a memory comparison is done of the data in the actual and the expected structure. Due to data alignment/padding, a boolean member that takes one byte in memory, will use 4 bytes in the structure. When a boolean is set, only one byte will be written, and the other three byte will be garbage. This results in a failed test. There is three ways to solve this. The first is to tell the compiler to pack the structures. The second option is to write zero values to the structure after initialization. The last option is to write a helper method and include this in the CMock configuration before generation of mock modules. It is also possible to ignore the method, if it is not important to test it.

When we have a mock module of the voltage comparator, it is possible to test that the loop waits for the measurement of voltage to finish, by setting exception and inject return values that makes the loop continue. If the loop is not waiting, the result will be that the function was called less times and the test fails.

3.7.2 Board LEDs

The board support package for the LEDs is used for the LEDs that are available on the development. The driver includes functionality for initialization, get LED status and to turn on and off LEDs.

Tests for these drivers were written with unit tests and mock objects. A mock module for GPIO and CMU was used, where expectations for all of the function call from the drivers was set for the tests. Unit tests were used to assert the return from the functions. Tests for all conditions for each of the methods were written. No problems were found when testing this driver.

3.8 Automated Build System

This section will look at automated building and testing of the system, both during development and testing, and as periodic testing of a complete repository. To be able to run automatic testing, we need to prepare the existing code for this. It will also be necessary to have a suitable test framework and the code need to be structured in a decent way.

3.8.1 Test preparation

To be able to automate the test activities, it will be necessary to do some modification to the code. Some of these might be permanent, while some changes are necessary to be able to run the tests on a host computer. A permanent change to improve the testability of the code is a manual task that can be done gradually when code is changed and tests are added. For functionality that is added we should focus on writing code that is testable. Many of the changes that are needed to make the code testable, can be automated by scripting. The script can be chosen to run when necessary or be run each time tests are run. To be able to generate mock modules automatic with CMock it is necessary to do modifications of the header files for the peripheral APIs. This was covered in section 3.5.

All the memory register to the peripheral device are set to specific memory addresses that are mapped to the device on the microcontroller. These addresses cannot be used while testing on the host computer. Since there are files that define the memory registers for each microcontroller, there will be several hundred lines to changes, which make it necessary to automate this task. Since the definitions of memory registers may be changed, it will be necessary to run this script each time the tests are run.

To be able to test the examples, several changes are needed. The first is to rename the main method under test. This change can easily be done by a script for all of the examples. Moving the loop of the function out of the main method is done in some of the examples, but not all. This is something that should be done to make it easier do to the testing, and make it possible to unit test the functionality that is done in the loop. This is a code change that must be done manually. Changing loops from a while to a do-while can be done by scripts before testing, it is an easy change that can be done by a simple parser, and the changes is only necessary for the test. Some example access memory registers directly on a given memory address. This will result in segmentation faults, and must be changed to either variables or use of a structure, when the tests are written for the example. Removing for example the delay function in the blink example should be done manual when creating tests, since this a feature that not occur in all examples, and other example can have equivalent features that need to be fixed during testing.

3.8.2 Ceedling

Ceedling is an automated building tool can be used with Unity and CMock [2]. Ceedling is a ruby gem, which uses Rake (Ruby Make) to perform the building [2]. Ceedling is just an extension of Rake to focus on testing and compilation of C projects. Ceedling is a good tool for supporting test-driven development since it has many options for automatic compilation and testing. By using Ceedling it is possible to write and run unit tests together with Unity and CMock, without having to create Makefiles, or create test runners and generate the mock modules with CMock.

The main functionality of Ceedling is the ability to compile and run tests, without having to specify which files that needs to be compiled and linked. A Ceedling project consists primarily of directories for source code and tests, and a configuration file. In the configuration file it is possible to specify options for the compiler and preprocessor that shall be used. In addition, the configuration holds the Unity and CMock configuration.

When Rake is used to test, it will look up the tests that are needed and create test runners for these tests. After this, compilation of all source file will be done. If a mock module is needed, this will be generated with CMock and compiled. The last steps are to link all the object files, run the tests and write the test results to the developer.

For testing, Ceedling can run all tests, run the tests in a given directory, test a specific file or only run tests for files that have changed since last time Ceedling was run. By having the ability to specify which test to run, it is good for use in test-driven development, where tests are run for each change. Testing all tests each time could result in a hour break between each change when using TDD.

3.8.3 Test structure

A good file structure is important to separate tests and source files. The source should be separated from test files, since it do not depend on functionality in the tests. The tests will consist of the test framework, the actual unit tests, mock modules and a test runner. The unit tests, mock objects and the framework should be stored separately.

A Ceedling project uses a simple structure, where all source- and header-files are stored in a source directory (src/). In addition, there is a test directory where all tests are stored. Ceedling will look through all subdirectories in both source and test folder and look for header, source and test files. This means that the structure in these directories does not matter. In the test directory the tests should be divided into unit tests and integration tests. The unit tests should also be divided according to the part of the software library that it tests.

Energy Micro already has their own system for compiling and testing their software. Changing this to use Ceedling would result in some work and there is also a risk that it could results in errors. A simple test, moving the source directory for the Energy Micro's software into a Ceedling project, was done. One example test and one peripheral API test was added to the project. The next step was to configure Ceedling, where configuration for CMock, Unity, the preprocessor and the compilator were done. There were only a two errors that were needed to fix before the tests was run:

1. Ceedling will generate the mock modules during compilation, the header-files that are used need to be prepared so they do not have any inline methods.
2. Some files had equal name, so the wrong file was chosen. The options for configuration of files with the same name were not looked into.

The biggest complexity in Energy Micro's software is that it has a peripheral API that shall support all of the microcontrollers. Many of the examples are also created so that they can be run on different development boards. The CMSIS peripheral support library takes care of definition of memory registers, availability of hardware and other definitions that are need for the specific microcontroller. The configurations that make the drivers work for the examples are defined in the board support package. The main problem with testing is how to be able to test examples for several development boards, without having to write duplicate tests for each of the development boards. Using the same tests to test the functionality in emlib for several microcontrollers is also a problem.

When compiling the peripheral support library, a definition of microcontroller is done in the microcontroller. To test this for all microcontrollers, the tests can be compiled and run several times, one run for each microcontroller. This can easily be scripted and done automatic, but there will be a problem with this solution: The microcontrollers do not have the same features and all tests are not necessary, and some of the asserted values will be different for each microcontroller. These problems have not been looked at in this thesis. But using the preprocessor to define which tests to run is possible. The same goes for the values that are asserted. Bit-field and masks are defined in the peripheral support library, and could also be used as values that are expected by the tests. Using definitions in the tests will make it harder for the developer to understand the values that are expected.

The definition of the microcontroller is done in the preprocessor for the examples projects. The same problems as with the peripheral APIs will also exist for the examples with software integration testing, but this will not be a problem when mock modules are used.

3.8.4 Automated test runner

With test-driven development, a developer will have to run the tests each time a change is done. By running the tests that often, it is important that tests can be run fast, at a maximum of one minute for all the tests that are run. It is therefore necessary to minimize the test that need to be run each time. The developer should at least run all the test for the module that is under development. It could also be necessary to test some the functionality that depends on the code.

By only testing a limited part of the software library under development, there is a need to do the rest of the testing elsewhere. A continuous integration server can be used for this, which can run tests, when code is committed to the software repository. If there are many commits to this repository, there might only be possible to run some integration tests here. To be able to test all, nightly jobs can be run, where all software is build and tested.

When testing the software library, a lot of test instrumentation code is added to the source code. This is code that only is used for testing purposes, and the code that shall not be shipped with the code to the customers of Energy Micro. Before release of code it is necessary to have scripts that go through all the source code to remove the test instrumentation. To ensure that the script is able to remove all of the instrumentation code, it is necessary to specify what kind of defines that is allowed for the test instrumentation.

3.9 Summary

This section has discussed testing of embedded systems, using Unity as a test harness and CMock as a tool for generation of mock modules from function prototypes in header-files. CMock generates mock modules that use Unity for assertions. Unity has the necessary functionality for doing assertions, and there exists plugins that can make Unity support fixtures. Since CMock does not accept header-files that consist of functions defined as static inline or by macros, it is necessary to modify these header-files before generation of the mock modules. Ceedling is an automated building tool, which was created to make Unity and CMock work together in a better way.

For software unit testing, the different parts of the software library have been viewed as different layers, as shown in Figure 2.2. When a part of the software package has been tested, all dependencies to other modules have been used as mock modules. By doing so, it is possible to test each of the unit in isolation, without concern about the functionality in the dependencies. To make it possible to create the mock modules automatic, a script was created to automatically remove inline functions in the header-file and change them to normal function definitions.

To check whether Unity and CMock could be used to test the functionality in Energy Micro's software, different peripheral APIs, drivers and examples was tested. All the parts were tested with use of mock modules and the peripherals were also tested by doing assertions with unit tests on the memory registers that was defined as structures. Testing an example on the development board was also done.

Making changes to the software to be able to test software can be time consuming. It may be necessary to create scripts that can remove test instrumentation before releasing code, and to change the definition of memory registers from fixed memory mapped addresses to pointer definition of the type structure. Scripts to modify header-files that will be used as input to CMock must also be used, due to the amount of static inline definition in the header-files. Automation of compilation and testing is necessary to be able to choose which tests to be run and to support agile development like test-driven development.

Most of the problems that occur during testing can be solved by adding test instrumentation that makes the preprocessor modifies the code when the code is under test. Macros can also be used to redefine behavior under test. In some cases it is also necessary to do some changes to the existing code. When writing to memory registers, it can be necessary to add read and write methods, to be able to create mock modules that can capture the data. In the example it is necessary to do some structural changes, so it is possible to run tests without going in an infinity loop.

CHAPTER 4 EXPERIMENT

This chapter contains the experiment that was performed to looks at use of unit testing and mock objects/modules to perform regression testing in an existing embedded system where functionality is added or modified. The unit testing will be done on the host computer, where mock modules are used as a hardware abstraction layer. The structure of the experiment follows the structure suggested in ”Experimentation in Software Engineering: An Introduction” [23].

4.1 Definition

This section forms the idea of using unit testing to perform regression testing when an existing system is refactored, modified or when functionality is added. In the experiment, the unit tests are created in advanced, so that the participant can verify that the implemented functionality behaves as expected. All the unit tests that are used, is integrated in mock modules.

4.1.1 Goal definition

The experiment is motivated by the Energy Micros case, where they have over hundred example project, where almost every project supports four different IDEs and the example may be used in different development boards. That an example runs on different development boards does not mean that they are equal, since the board layouts are different. This will result in over thousand binaries, which is almost impossible to test manual by deploying to development board and then run regression or system tests. Important to look at efficiency by using unit tests to do regression testing and see if the number of errors is reduced when using unit testing to perform regression testing after modification or refactoring of code without tests.

When unit test is used without access to hardware, there are several approaches that can be used; some of them are listed below:

- Use mock objects/modules as a hardware abstraction layer.

- Setup all memory registers as variables, and perform assertions to these.
- Run the tests in a hardware simulator.

This experiment use mock modules as a hardware abstraction layer, where expectations is set, assertions are done inside the mock module, and return values is injected. When using these methods, the tester does not have full control over functionality that is done by the hardware, but it can verify correct calls to the hardware abstraction layer, and assert that the calls are done with correct arguments.

The objects of the study is existing source code of an embedded system. The participants will write code to complete the requirements that are specified for the task. Since refactoring is difficult to measure, the participants will add functionality to the system or modify the code, to change the behavior of the system. Some of the tasks will also be tasks that are not working, and the participants should make them work by adding the missing parts. For the experiment there will be five tasks, described briefly below:

1. This task is a simple application that counts down from a given value to zero. It uses the segment LCD to display the count value, and uses the LEDs to indicate that the counter has reached zero. The participants will need to implement the initialization of the segment LCD and the LEDs, and then implement the behavior when the counter reaches zero.
2. The application uses the push buttons on the development board to turn on and off the LEDs. In addition the total number of clicks is written to the segment LCD. The participants will implement the interrupt handlers for the buttons, so that the LEDs are set or cleared and the total numbers of clicks are written to the display, when a push button is clicked.
3. This task uses the push buttons and the segmented LCD. The segment LCD will display a binary string, and the decimal value of this string. The buttons will be used to increment and invert the value of the binary string. The participant will implement the behavior of the interrupt handler and write the binary string to the segment LCD.
4. This is a simple application that displays a string on the segment LCD. The task for the participants is to modify the functionality so that the user can read the whole string on the display. To be able to do so, the string will need to rotate from the start to the end on the screen.
5. This application uses the capacitive touch slider on the development board. It reads the value from the capacitive sense interface and displays the value of the touch slider on the segment LCD. In this task the experiment participants will do the initialization and setup of the capacitive sense system. In addition they will modify the functionality so that the segment LCD is turned off when no value is read from the touch slider.

Object of study The object of study is the source code of existing embedded systems, and how this system can be tested in an efficient way during refactoring, modification of the source code or when functionality is added, to reduce the amount of errors before the software is deployed to hardware.

Purpose The purpose of the experiment is to see if use of unit-testing during development of embedded system can reduce the number of errors and see if the developers will be more efficient when using unit testing instead of system and regression testing after deployment of code to the embedded system. If the reduction of errors is sufficient it may not be necessary to test all binaries on hardware.

Quality focus The main effects that are studied in the experiment are the reduction of errors under development on embedded system, by using unit tests and mock modules/objects. If numbers of errors are reduced it may also reduce the time used on testing.

Perspective The perspective is the developer's view, since they are doing the testing and therefore want an efficient way of doing testing, which will reduce the number of errors in the software. Outside the experiment both project managers and customers may be interested in the reduction of errors before the source code is delivered, since they want software without errors.

Context The environment of the experiment is a computer lab that consists of a computer and an Energy Micro EFM32GG STK3700 development board. The participants in the experiment will modify an existing source code for five simple examples, which will be provided. The participants will be students with basic knowledge of programming in C.

4.1.2 Summary of definition

Analyze using mock modules as a method for testing an embedded system for the purpose of regression testing with unit tests with respect to reduce number of errors and efficiency from the point of view of the developer in the context of a computer lab.

4.2 Planning

This section provides the planning part of the experiment, which uses the definition from the previous section. The planning consists of context selection, hypothesis formulation, variable selection, the selection of subjects, experiment design, instrumentation and validity evaluation.

4.2.1 Context selection

The experiment is an off-line context since it uses tasks that are created in advance for this experiment. Experiment is not done in a professional organization, but in a lab.

The participants in the experiment will be students that know the basics of programming in C.

The code used in the experiment is not created for any real project, so the experiment will be a toy problem.

The problem is specific because the focus is to use unit testing with use of mock modules for regression testing of embedded software.

4.2.2 Hypothesis formulation

The subjects in the experiment will be divided in to groups, one that uses unit test, the number of errors for this group is expressed as μ_x. The number of errors for the other group, is expressed as μ_y, this group not will have access to unit tests. In this experiment μ measures the number of errors found during testing on development board. The number of errors found during testing can be compared by a ratio scale. The number of errors for each group is the number of errors found with system tests after the code is deployed to the development board.

The null hypothesis is that there is no difference in number of errors found during testing between the group that uses unit test and the group that not uses unit tests.

$$H_0 : \mu_x = \mu_y \tag{4.1}$$

The alternative hypothesis H_1 defines that the testing on the development board resulted in fewer errors for the group that used unit tests.

$$H_1 : \mu_x < \mu_y \tag{4.2}$$

4.2.3 Variables selection

Independent variables The independent variables are the students that will perform the experiment, and the difficulty for each of the tasks the participants will do in the experiment. The independent variable that will be studied is the development process. In this case this will be use of unit testing or without use of unit-tests.

Dependent variables The dependent variables in the experiment will be number of errors that were discovered when testing on the embedded system. The number of errors are found when system tests are executed after the code is uploaded to the embedded system.

4.2.4 Selection of subjects

The subjects that are used in the experiment will be students. The students have some experience with the C programming language, and most of the students should have some experience with programming for embedded systems. The sampling used in the experiment is convenience sampling.

The tasks in the experiment do not require deep understanding of the C programming language, but they should know the basic syntax of the language and the use of pointers. The experience with programming on embedded system is needed to be able to finish the experiment in a reasonable time, since the participants will need to use documentation for APIs, reference guides, and be able to download the compiled binary to the development board.

4.2.5 Experiment design

The factor for this experiment will be the development process. The first will be a group that uses unit testing during development on the tasks. The second group will have to implement the functionality without being able to use unit tests.

The experiment will thus have one factor with two treatments, and use a completely randomized design. The method of analysis will be the t-test.

Randomization The subjects for the experiment are not chosen randomly, since they need experience with the C programming language and experience with programming on embedded systems. The subjects will be assigned to one of the treatments randomly.

Blocking Due to the different amount of experience with programming on embedded systems, it will be necessary to put the subjects into blocks and then divided them into different treatments. This is necessary to eliminate the effect of different experience with programming on embedded systems.

Balancing Each of the treatment in the experiment will have an equal number of subjects.

Standard design types The design of the experiment is one factor with two treatments, where the factor is the development process with use of unit testing, and the treatment is using unit tests or not. The number of error is the important dependent variable, which can be measured on an interval scale. In this case a t-test will be used for analysis of the data.

4.2.6 Instrumentation

The participants will be given five tasks that are the objects of the study. They will add functionality and modify the source code for these tasks. The group that has access to tests can run these tests and modify the source code until tests pass or they give up. When the groups have downloaded the code to the development board they will not have access to modify the source code.

The participants will be given guidelines that describe the tasks and the steps in each of the tasks. The documentation will consist of instructions to upload code to microcontroller, and for each task there will be requirements, instructions and test cases. In addition we will hand out necessary documentation of the software library.

When the system is running on the development board, the participants will perform the tests cases that are given in the guidelines and documentation to the experiment. The total number of errors, which is the dependent variable, will be the number of test cases which failed. A failing test case that fails because of an error in one of the other test cases will not be counted as an error.

4.2.7 Validity evaluation

The threats to the validity follow below:

- The participants get bored if the experiment lasts too long. We do not know how long the experiment will last. We have tried to reduce the amount of work that is necessary for each task, but each task is quite repeatable since each task will require deployment of code to embedded system and it needs to execute test cases.
- The task is too easy so it will not create any difference between the two groups. The tasks are created easy so it can be possible to accomplish the experiment in 1-2 hours, and there should be some tricky details that may produce errors for the developers.
- Not enough participants that participate in the experiment will result in weak statistics result and make it difficult to make any conclusion of the results from the statistically analysis.
- One of the groups will use mock modules with unit tests. By looking at the tests code the participants can use this as a guideline for how to implement the functionality for the task. The only way to prevent this is to ask the participant to not look at the test code, but use the tests to find errors in the software.
- Due to limited space on the computer lab, not all participants can do the experiment at the same time. This may be a problem since the last participant can be informed about tasks in the experiment. It is probably only a few or none that have knowledge of the Energy Micro microcontrollers, and students will probably not try to find solution before they meet up on the experiment, this should not be a problem.
- If the documentation and guidelines for the experiment are not good enough, it will result in problems for the participants, which can influence the results, since they not are able to carry out the experiment. This may be prevented if we test the experiment before the experiment is carried out by the subjects.

4.3 Operation

The operation of the experiment uses the design and definition to carry out the experiment. This section consists of the preparation, execution and data validation.

4.3.1 Preperation

The participants were aware of that the experiment would look at unit testing on embedded systems. They did not know how the result was measured.

Finding participants that had experience with C and microcontrollers was difficult. The experiment ended up with 16 participants from XCOM14, where only half of the participants had experience with C/C++. The experiment was made simpler, to minimize the use of pointer/data structures that differ from Java programming. Each of the participants was paid 500 NOK.

Each of the participants got the guidelines and documentation at the start of the experiment. The documentation included documentation for the drivers and APIs they needed to use to implement the necessary functionality. The guidelines consisted of an introduction, instructions of how to download the necessary source code, tools and compiler.

In addition it included instructions for how to upload code to the hardware. The most important part of the guidelines is the instructions of how to perform the five tasks. The tasks consist of requirements, instructions and description of how to implement this. The experiment guidelines is found in Appendix A.

Each of the participants needs to download the necessary package that included tests, source code, compiler and tools to upload binaries to the development board. The source package and the guidelines were created in versions for the group with tests and the group without tests. The guidelines in Appendix A are for the participants with tests.

Before the experiment was executed with the 16 participants, one participant tested the experiment to check that it was easy to understand, how long time it took to finish the experiment for a participant and that everything worked. The participant had about the same experience as the participants, the experiment took over 3 hours and we decided to skip test cases and upload to hardware. The result of this participant was included with the results.

4.3.2 Execution

The experiment was performed on 2-3 May by four groups that consisted of four participants. Since half of the group had no experience with the C programming language, they got help with syntax and a few hints on how they could solve the tasks. The participants used in average 2.5 hours to finish the experiment, where the fastest was finished in less than 2 hours, and the slowest used almost three and a half hour. The participants were observed during the experiment to look at how they solved the tasks and what kind of problem they experienced with the unit tests.

Before the execution of the experiment, all necessary source code, tools and compiler was download and prepared. This was done so that they could use most of the time on implementing the functionality given by the tasks. The participants did not use hardware and execution of test cases. This was necessary to reduce the time used by the participants.

After the participants had finished the experiment, all the tasks were compiled and upload to the development board, before the test cases was executed. This was not done by the participants. The test cases for the tasks is found in Appendix B.

4.3.3 Data validation

Data was collected from 17 participants, where 8 executed the experiment without unit test and 9 finished it with unit tests.

Three of the participants had little experience with programming and had problems with many of the tasks. Two of these participants executed the experiment with unit tests, and ended with 8 errors each, which is twice the mean value. The participant without tests ended up with the same number of errors as the mean value for the participants without tests.

The results from the experiment are found in the next section.

4.4 Analysis & Interpretation

This section uses the experimental data from the execution of the experiment to present these data and make conclusion that is made from the results.

4.4.1 Descriptive statistics

This part presents the results from tests that were executed after the participants have finished the experiment. The presentation consists of a boxplot of the data and a graph that shows the outliers.

Boxplot

The boxplot in Figure 4.1 shows the number of errors for each of the participants seen in Table 4.1. Figure 4.2 shows the distribution of errors for participants with and without tests. In the last figure the x-axis shows the number of errors, and y-axis shows the number of participants that had the given amount of errors. This graph is used to identify the outliers.

Table 4.1: Errors per Participant

With tests	Without tests
8	2
7	2
8	4
4	8
9	3
7	2
8	4
8	8
	3

Results

The total results for each of the test cases can be seen in Table 4.2.

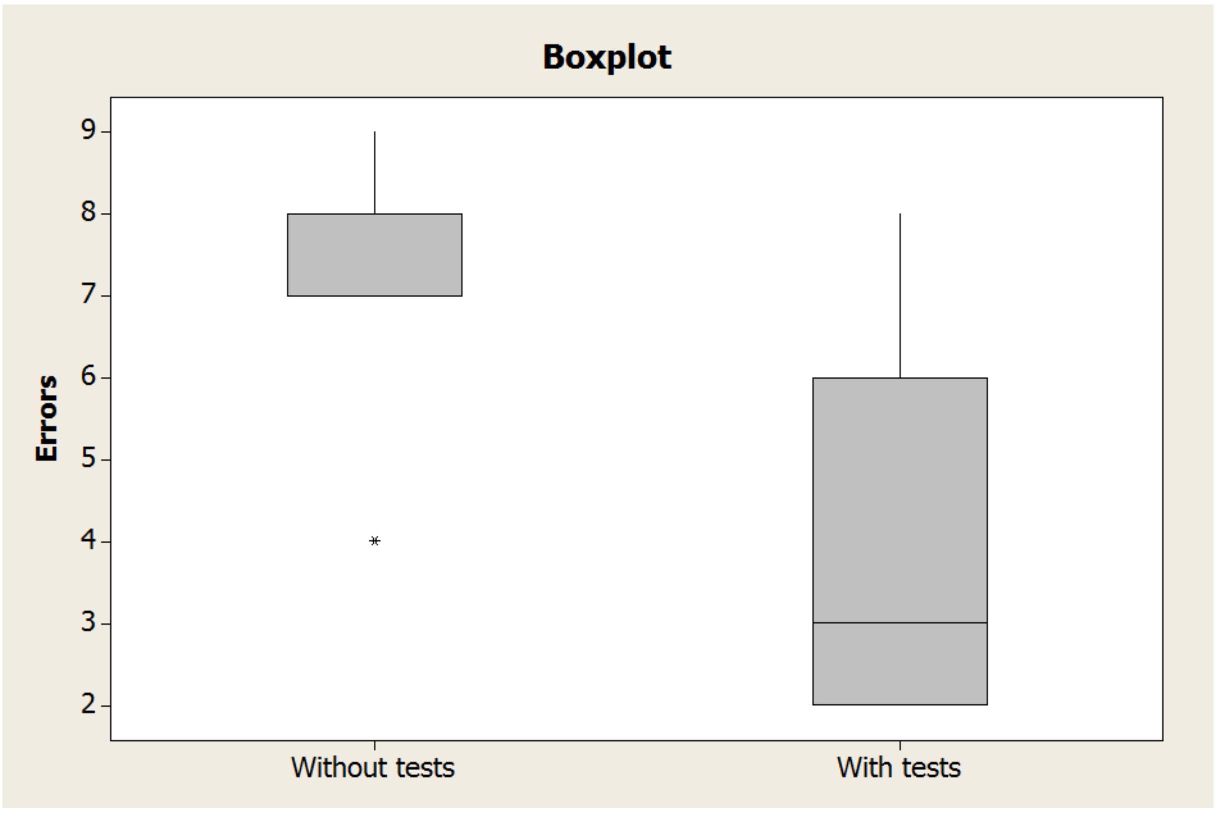


Figure 4.1: Boxplot of Test Results

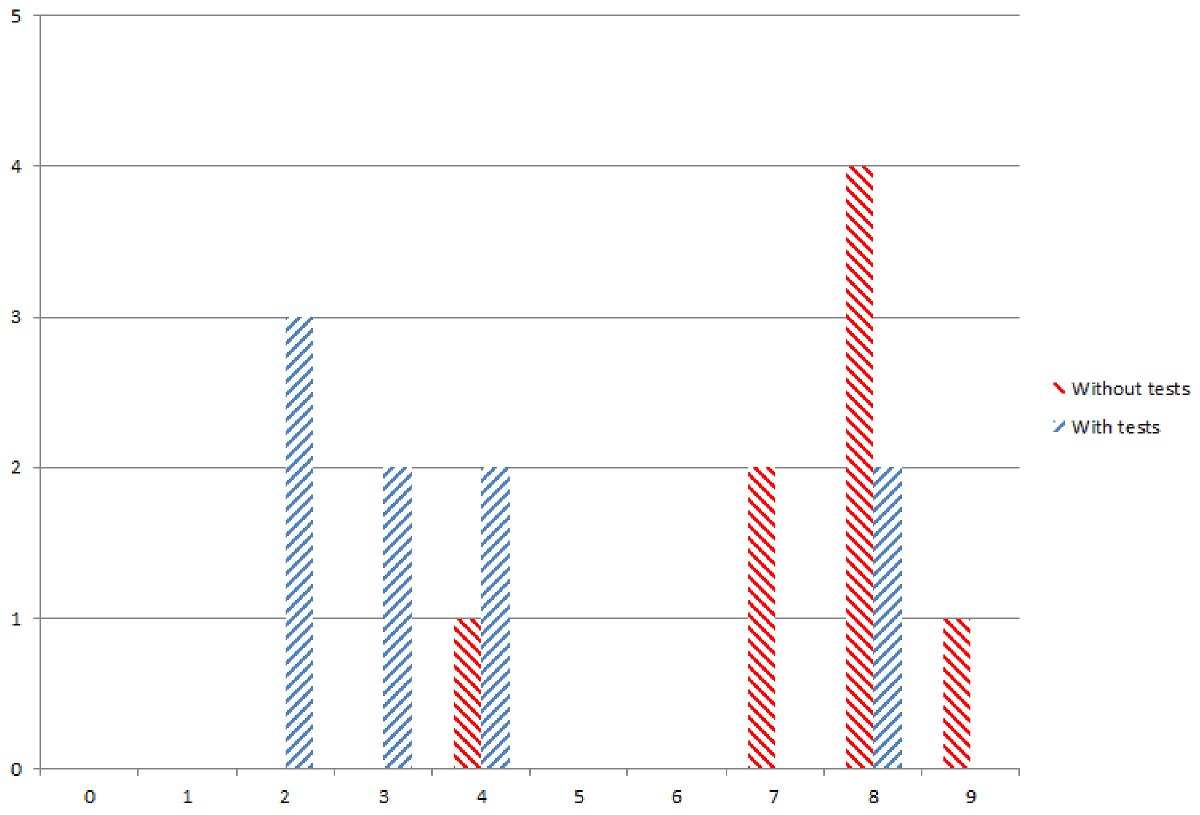


Figure 4.2: Distribution of Errors

Table 4.2: Errors per Test Case

Test case	Errors		
	With tests	Without tests	Total
Existing functionality is working	0	0	0
Segment LCD initalized	0	0	0
BSP LEDs initialized	0	1	1
LEDs blink alternating	2	3	5
RESET Centralized	0	7	7
Task 1	**2**	**11**	**13**
Existing functionality is working	0	0	0
Buttons is working	1	1	2
LEDs updated on click	1	2	3
Falling edge	1	0	1
Interrupts is cleared	1	4	5
Display updated on click	1	1	2
Task 2	**5**	**8**	**13**
Existing functionality is working	1	0	1
Value incremented on right click	1	0	1
Value inverted on left click	1	2	3
Number is correctly updated on click	2	0	2
Text is correctly updated on click	7	7	14
Task 3	**12**	**9**	**21**
Existing functionality is working	0	0	0
Value is written to segment LCD	1	3	4
Pattern in requirement is correct	4	8	12
Rotation restarts when reach end	5	4	9
Task 4	**10**	**15**	**25**
Existing functionality is working	0	0	0
Caplesense is initialized	2	2	4
Segment Ring correct	4	6	10
Segment Number correct	0	5	5
Display empty on no input	1	3	4
Task 5	**7**	**16**	**23**
Total	**36**	**59**	**95**

4.4.2 Observations & results

Observations and results description for the participants is found below; they are divided into sections for the group without tests and the group with tests.

Participants without tests

This first task was easy for the participants, since they were told what to do in the instructions. With the test cases, 11 errors were found. The most frequent errors were that text was not centered as defined in the last requirement, and that the LEDs did not behave as required. Results for each of the requirements are found below:

TC1-1: No errors were found by any of the participants, since there was no need to modify any code in this section.

TC1-2: No errors were found during testing.

TC1-3: One error, the participant forgot to initialize the LEDs.

TC1-4: A total of three errors. The main failure is that they did not use the LED toggle to switch state on the LEDs. Some of the participants tried to add some logic to first check if the LED was on then cleared or set it.

TC1-5: Seven failures were found. Almost all had written "RESET" left aligned.

The test cases in task 2 resulted in eight errors for the participants without tests. Only a few errors in this task. It was easy for the participants to do, since they are told exactly what they needed to do. Description of the errors follows below:

TC2-1: No errors found.

TC2-2: One error found during testing, due to wrong parameters set in the pin setup.

TC2-3: Two errors found since the functions to update LEDs never were called.

TC2-4: No errors found.

TC2-5: A total of four errors, the main reason were that the participants did not know how to calculate the correct pin.

TC2-6: One error was found, the reasons for errors was that the functions was not called in the interrupt handler.

Nine errors were found in the third task for participants without tests. Just a few errors before the binary string should have been written to the segment LCD. There were many failures on the last test case, since many gave up this task. Some errors were due to writing 0 and 1 to the char array instead of ’0’ and ’1’.

TC3-1: No errors found.

TC3-2: No errors found.

TC3-3: Found two errors here, one task was not done and the other set the value incorrectly.

TC3-4: No errors found.

TC3-5: Very difficult task for the participants, only one made this test pass. There were many different approaches to solve the problem.

In the fourth task, 15 errors were found. Most of the errors are that the string is not displayed correct. The errors occur when the string is displayed wrongly at the start of the rotation, or at the end when the rotations shall start from the beginning. Without tests, it is difficult to verify that the string is written correctly.

TC4-1: None broke the existing functionality.

TC4-2: Three participants forgot to write the result to the segment LCD.

TC4-3: A total of eight errors here. There was many different errors, most of them displayed the string wrong in the end.

TC4-4: Four errors in this task. The main error is that the string never is set to begin at the start again, which may result in a segmentation fault.

There were 16 errors in the last task. The main failures are that not all requirements was implemented, and problems turning the segment rings on and off correctly.

TC5-1: No errors found.

TC5-2: Two errors found in initialization and set-up. The documentation is not good on describing what each of the functions does.

TC5-3: A total of six errors. Most of the errors are that the indicators are not turned off when the value becomes lower and that wrong indicators are turned on/off.

TC5-4: Five errors for this test case. Either they forgot to write number or they modified the value before it was written to the display.

TC5-5: For these tests there were three errors, most of them because they forgot to turn off either the ring or the number segment.

Participants with tests

The first task was easy and the description told the participants what to do. Two errors were found in the implementation by the participants with unit tests. Using the tests as in TDD is possible, since there are only a few errors the tests will give feedback on. The difference from TDD is that the participants did not need to write the tests. Starting running the tests from the beginning will result in two failed test cases for methods that were not called. In addition, there are three tests that are passing, which covers the existing functionality. The unit tests worked good for the participants for this task, since the feedback that was given from the test result was easy to understand. Observations for each of the test cases follow below:

TC1-1: No errors were found by any of the participants. Tests for the existing functionality were included.

TC1-2: No errors were found during testing. Mock modules verified that this was done.

TC1-3: No errors, the test checked that this was done.

TC1-4: A total of two errors. The main failure is that they did not use the LED toggle to switch state on the LEDs. Some of the participants tried to add some logic to first check if the LED was on then cleared or set. It was difficult to understand why the test cases were failing with a message that a function was called more times than expected. It is possible that the solution could have worked, but it could not be checked by the tests since the test expected that the problem was solved in a different way.

TC1-5: The participants had no errors. Many correct this error by running unit tests. It was easy to understand since the tests will report with what string it expects and what it actually was.

Five errors for the users with test were found in the second task. One of the participants had 4 errors because of little understanding of how to solve the task. It was easy for the participants to do, since they are told exactly what they need to do. Users with unit test were able to correct errors with wrong parameters in pin setup and correct pin to clear for an interrupt. The output from the tests can be difficult to understand, since many function calls are done in each test. This will result in that almost all test say that a function was not called as expected, and the error message for this error is printed first since the mock modules is verified first. The tests worked well for verifying that the correct parameters were given to the function that was called. The observations follow below:

TC2-1: No errors found.

TC2-2: One error found during testing, due to wrong parameters set in the pin setup.

TC2-3: One error found since the functions to update LEDs never were called.

TC2-4: One error found. This error occurred since interrupt configuration was not done.

TC2-5: One error, the main reason was that the participants did not know how to calculate the correct pin. Participants with unit tests could find out and verify that correct value was set.

TC2-6: One error was found, the reasons for errors was that the functions was not called in the interrupt handler.

In the third task 12 errors there were found, which are three errors more than the group without unit tests. A few errors were found before the binary string should have been written to the segment LCD. There were many failures on the last test case, since many gave up this task, and some errors were due to writing 0 and 1 to the char array instead of '0' and '1'. All of the tests run the "loop", which results in calls to the segment LCD. This make all tests fail if the number or text is not written correctly to the display. The best way to solve this is to declare variables as external in the header-file and use unit tests to assert the value, in addition to mock modules. One of the participants had four errors on this task.

TC3-1: One error found where the participant used a break statement in the infinity loop, the test cases was not able to detect this. This is also difficult to test with a unit test, since the tests cannot run to an infinitely long time.

TC3-2: One error since the increment was put inside an if expression that never was true, the error is discovered by the unit test.

TC3-3: One error found here. The functionality was not implemented.

TC3-4: Two errors found. The value was modified inside the main loop in an approach to write the binary string to the text segment. The error was discovered by test, but never fixed.

TC3-5: Difficult task for the participants, only two participants made this test pass. There were many approaches to solve the problem, for example greatest common divisor and comparing each bit. Unit test do not help when participants do not know how to solve the task.

In the fourth task, ten errors were found. Two things are tested with mocks here: 1. The initialization is done correctly. 2. The correct value is written to the display for each iteration in the loop. It can be difficult to distinguish between the start and the end of the pattern in the unit tests. Some of the participants took advantage of using unit tests to guide them to the correct result. Many of these errors with unit tests could have been corrected by trying and failing. The experiment also shows that it is difficult to test logic by using mock modules, unit test assertions should be used instead.

TC4-1: None broke the existing functionality.

TC4-2: One participant forgot to write the result to the segment LCD.

TC4-3: Four errors were found. Many different errors, most of them displayed the string wrong in the end of the rotation.

TC4-4: Five errors in this task, this is more errors than for the group without tests. The reason for this is unknown, it could either be random or that the participants had problems to solve the previous task, and did not get this feature tested. The main error is that the string never is set to begin at the start again, which may result in a segmentation fault. With unit test it is difficult to understand if the error is detected at the beginning or the end of the string, since the same string is expected. Another failure was that the whole string was rotated inside the loop without delay, the participant was sure that the result was correct, and that his implementation was different from what the test expected of calls to the modules.

There were seven failures for the last task. There are many ways to solve this problem, even if the unit tests fails, since there are many approaches and functions that can be used to meet the requirements. The unit tests uses one approach, and has expectations for which functions that need to be called. This can be confusing for the participants. The tests depend on the loop, since the data are retrieved by polling. It was possible to get the test passing and the test cases failing, since functions were called in the wrong order. It is possible to configure CMock to generate mock modules that verify that functions calls are done in correct order, but it will probably be difficult to use in practice. Updating the segment ring needs eight method calls to the driver. When so many calls are done in a test, it is difficult to use the output the failing test gives.

TC5-1: No errors found.

TC5-2: Two errors found on initialization and setup. Documentation is difficult to understand. One of the unit tests passed, but failed anyway, since the functions were called in the wrong order.

TC5-3: Four errors found. The output from unit testing is difficult to understand, since there are many functions call to the ring indicators. Most of the errors are that the indicators are not turned off when the value becomes lower and that wrong indicators are turned on/off.

TC5-4: No errors found.

TC5-5: One error found, forget to turn off the ring indicators. This participant had also an error on test case 3, and could therefore not see the error.

4.4.3 Data reduction

When looking on Figure 4.2, we see three outliers. The first is the participant without tests that only had a total of four errors on the tasks. The two other outliers performed the experiment with tests and resulted in 8 errors, which is twice as much as the other with tests.

The participant with only four errors without using tests was the most experience programmer, with some C++ knowledge. We do however, not see any reasons to remove this participant from the analysis.

The two outliers with tests did not have much experience with programming. Half of the errors for these participants were by failing completely on one of the tasks. The first had four errors on task 2, where there were totally five errors for all the nine participants without tests. The second outlier had four errors on task 3, by neglecting test case 5, this is more than half the errors for all participants with tests. Since they have about an average numbers of errors of the other tasks, these outliers are not removed from the analysis, but the errors on task 2 and 3 should be noted.

4.4.4 Hypothesis testing

The hypothesis that developers using unit test based on mock modules results in fewer errors than developers without any tests is calculated with t-test. Below follows the calculation of the t-value.

$$n_x = 8 \tag{4.3}$$

$$n_y = 9 \tag{4.4}$$

$$\bar{x} = \frac{\sum_{i=1}^{n} x_i}{n} = 7.375 \tag{4.5}$$

$$\bar{y} = \frac{\sum_{i=1}^{n} y_i}{n} = 4.000 \tag{4.6}$$

$$s_x = \sqrt{\frac{\sum_{i=1}^{n}(x_i - \bar{x})^2}{n-1}} = 1.506 \tag{4.7}$$

$$s_y = \sqrt{\frac{\sum_{i=1}^{n}(y_i - \bar{y})^2}{n-1}} = 2.398 \tag{4.8}$$

$$t = \frac{\bar{x} - \bar{y}}{\sqrt{s_x^2/n_x + s_y^2/n_y}} = 3.514 \tag{4.9}$$

Since variance is not equal for the two samples, degrees of freedom are calculated with:

$$u = \frac{s_y^2}{s_x^2} = 2.535 \tag{4.10}$$

$$df = \frac{(\frac{1}{n_x} + \frac{u}{n_y})^2}{\frac{1}{n_x^2(n_x-1)} + \frac{u^2}{n_y^2(n_y-1)}} = 13.61 = 13 \tag{4.11}$$

Degrees of freedom is rounded down to 13. By look up the t-value and degrees of freedom, the p-value is 0.004. The statistical results are shown in Table 4.3.

Table 4.3: T-test Results

	N	**Mean**	**StDev**
Without tests	8	7,375	1,506
With tests	9	4,000	2,398
	df	**t-value**	**p-value**
t-test	13	3.514	0.004

The hypothesis H_0 is rejected since the difference with and without unit tests is significant, with a p-value as low as 0.4%. The reasons for this is that developers with unit test are able to actually test and correct errors before testing software on hardware. The feedback given for correct functions calls is also valuable.

4.4.5 Threats

There are three things that threaten the results from being applied to the real world. That is the experience of the participants in the experiment, the complexity of the tasks and that it was simulation.

The experience with programming among the participants was low, since half of the group had experience with programming in C or C++. Using a group with higher experience and good knowledge would reduce the number of errors to near zero, and the group with tests should have zero errors, but they could also have errors. In the tasks there were found two types of errors that could not be found by the test. The first of these errors was that function calls where done in wrong order. The second error was that a break statement was used inside the infinity loop, which could not be discovered by the tests, since they only run the loop on time.

The experiment was simulated; it was not performed on a real problem, but for a few tasks that was created for the experiment. Testing this on a real problem would look on other aspects of testing with unit tests and mock modules since the problems become more complex.

The complexity of the tasks is low, since the participants has little experience with programming. This makes it difficult to test more advanced functionality and applications with much more code. How well the testing of an application with over thousand lines of code would work is difficult to say after this experiment, but it is reasonable to believe that errors also will be found in a complex system. In a more complex system, it is natural to use more unit tests to test the logic, and the complexity would also need more test instrumentation.

Since the experiment is a simulation of a real problem and it was aimed to last two hours for each participants, it may be difficult to compare the experiment to a real situation. But even in a real situation, the mock modules will work in the same way, and assert the number of calls and the parameters of these calls. In a real problem there will also be more experienced developers that will be able to solve all tasks, so the significant result may be lower.

4.5 Summary

The experiment with comparing not using tests and testing with mock modules was done by 17 participants. Using mock modules to unit test the software was significantly better than not using tests. It was difficult to find participants to experiment, and second grade student was used, which have limited experience with programming and programming in C.

There is a difference in where the participants with tests have errors. In the first two tasks, where the users had clear requirements to implement and good guidelines on how to solve it, there were only a few errors. On the three last tasks there is more errors, the participants need to implement some logic, and the participants with little experience has problems implementing the logic. It is also difficult to understands the results from the mock objects here. It will probably be better to use unit tests. The structure from the examples provided by Energy Micro was used for the tasks, but was changed as little as possible to be able to write tests. Structuring the code even better would make it easier to also add unit tests. It is also possible that using a stub function also would help, if functions with logic are called many times.

The mock modules verify that a function is called, and that the correct parameters are passed to the method. There were found errors that not could be discovered by tests, since the methods was called in the wrong order. This could be solved by also verifying the order of calls, but might results in that there will be more difficult to change code and write tests. Understanding the results from the mock modules is difficult when we have many method calls to the same method in a function under test. When many mock modules is used in the same test, the test will fail and return the error of the first mock module that fail, which may give unexpected messages as a test results, which is difficult to understand. This can in many cases be solved by having a better structure where a function has fewer dependencies to other modules.

The result from the experiment is significant at the 1% level, and the mock modules will find errors and verify the correct behavior also in real problems. Testing of logic will be tested better with a better structure of the code and when using unit test to assert this part. Using mock modules to verify behavior, will work as regression testing to check that correct calls still are done to the dependencies.

Further work will be to use mock modules in a real project that is developed by experienced developers. It can also be important to look at how well use of mock modules will work when changes are done in the modules that depend on hardware, where there are only used mock modules.

CHAPTER 5 DISCUSSION

The section discusses the results from the previous chapters and does an evaluation of this. The chapter does also answer the goals that were given in the first chapter of the thesis.

5.1 Experiment

The result from the statistical analysis in the experiment showed that use of mock modules was significant better than not performing any tests of the software. The result was not a surprise since testing will reduce the amount of errors in software. The average number of errors for participants with tests was 4 and the average number of errors was over 7 for participants without tests. There were a few outliers that were included in the analysis, but without these the difference would have been bigger. The outliers come from the difference in experience by the participants, and the average participant had low experience in programming in general and with programming in C. The low experience among the participants made it necessary to simplify the experiment. The low experience and the simple task are probably the biggest threats to validity of the experiment.

Another threat to validity is that the experiment is only a simulation, and is not based on a real problem. If the experiment was run with experienced developers with knowledge about the Energy Micro microcontrollers, the differences between in errors would probably be smaller and the average of errors would be much lower. It would also be better to try the experiment in development of a real example. Because of the significant results in the experiment, it is likely that use of mock modules would reduce the number of errors even with experienced developers in a real software project. Since the problems and tasks are only simulation, they are still similar to the examples that are created for the STK3700 start kit from Energy Micro. The main difference is that the examples used in the experiment are simpler.

The first research question in thesis is “Is it possible to reduce the number of errors during development of new features, modification and refactoring of existing functionality by using mock modules to verify the desired behavior of the system?” From the statistical analysis from the experiment, the results of this question is that it is possible to reduce the number of errors when developing new features or modifying existing functionality. The experiment did not look at refactoring, since this is complex and require code review to see that it actually was changed and run the tests to see if it is working. By injection an error into an example that has good unit test coverage and mock modules, it is possible to detect this error by running the tests, so using mock modules would probably reduce the number of errors in refactoring too.

In addition to the statistical analysis we did some observations during the experiment. The main problem with test results, is that it can be difficult to understand the feedback given, especially when many calls are done to the same methods, which will results in an message saying that a method was called to few or to many times. The same problem occur when many of the tests involves the same modules, where the tests will verify the same mock modules first, and all tests will fail because of these, without possibility to see if other features are working. Testing logic in the application with mock modules also makes it difficult to understand results, so testing logic should by unit test assertions instead of mock modules, where mock modules is used to solve some of the dependencies. From the test results it is possible to see that the number of errors is much higher for participants with tests on task 3 and 4, which are the tasks with most logic. Most of the problems above can be solved by making the code more testable, by using more functions, where each function has fewer dependencies.

Two errors was also found that not was discovered by the unit tests, one of these could have been found by changing the configuration of CMock, the other could not have been found by the unit tests. But software unit testing is only one necessary test activity, and some errors will be found in other test activities. It is impossible to be able to detect all kind of errors in software unit testing. Developers will also create errors when writing errors while writing unit tests.

Advice 1: Use of mock modules reduces the number of errors in development and refactoring of embedded software.

Advice 2: It is difficult to test logic with mock modules, so variables in the examples should be made accessible for the tests, so that the variables can be checked with unit tests assertions instead.

Advice 3: A good code structure will make it easier to write tests for the software, the developers should therefore focus on testability.

Advice 4: The units that are tested should be as small as possible, because dependencies to many mock modules may result in feedback from the tests that are difficult to understand.

5.2 Mock Modules

If Energy Micro wants to do software unit testing of their software library, mock modules will be an important part of the solution. Mock modules will be important for three reasons. The first is that it does internal assertions of parameters, and assertions against the expectations that are set for the Mock modules can also be used to inject return values, when needed. By doing this, we can test that a unit calls the correct methods and that the dependencies are removed from the unit during test. The second reason is that hardware is not available during tests, and by using mock modules these can be used instead of the real modules that need to access hardware, and return values from hardware can be injected into the mock modules when setting expectations. The third reason is what we will be able to capture data that are written to registers that cannot be read and asserted, but the expectations can be set to verify all calls.

Energy Micro will need to create mock modules for all the peripheral APIs, the board support packages and the device drivers. This is needed to be able to perform software unit testing, where the units are as small as possible. The mock modules will be important in testing of the examples, where the experiment showed that the number of errors was reduced with use of mock modules during testing. During software integration testing and testing of peripheral APIs there is also need for some mock modules for methods that read or write to memory registers, to be able to capture the data and do assertions on data that are read or written to hardware.

CMock is a great tool for generating mock modules, and automates much of the work with creation of test doubles. A few problems have been found with CMock: the most important one is probably the support for inline methods and functions defined in macros. This can easily be solved by writing a script that adds non-inline methods during test. CMock also lacks support for doing assertions on structures and custom types. Currently these assertions are done by comparing the memory of the actual and expected structure. CMock has announced that CMock will support structures and custom types in later version, but currently there is no development in work. CMock is also open-source project, so participation in development is possible if features are needed.

Observations from the experiment showed that too many dependencies also will results in many mock expectations, which can be difficult for the developer to interpreted. A typical structure of an example is a main method, which consists of set up of the microcontroller, and a loop that contains most of the functionality. In addition there may be a few methods. To improve the testability in the examples, it is necessary to split the functionality into methods to reduce the responsibility for each of the existing methods. Many of the memory registers is accessed directly from a function. When these read or write to memory registers that cannot be read by the test harness it is possible to verify that the correct values was read or written to the registers. To be able to test these it might be necessary to create inline methods that access the memory registers, with this method it is possible to create a mock module that can capture the data that are written, or to inject return values for data that is read.

Advice 5: To be able test the examples, it is necessary to create mock modules of emlib, drivers and board support package.

Advice 6: It is necessary to create script that modifies the inline functions in the header-files, so it will be possible to generate mock modules with CMock.

Advice 7: To improve testability, it is necessary to split the functionality into methods to reduce the responsibility for each of the existing methods.

Advice 8: It may be necessary to create inline methods to read and write to registers, so mock modules can be create the capture the calls and inject return values to the register.

5.3 Embedded Testing

Unity and CMock works well together when testing embedded software. Where CMock is used for generation of mock modules that are used for testing, all assertions in CMock are with Unity. Unity has a wide range of different assertions that can be used, and many of these are created for the needs of an embedded systems. The main problem with CMock is that it does not accept inline methods and functions defined in macros, but this can be fixed by preparing the header-files with a script before generation of mock modules. Ceedling is also a good tool. This is an automated building system that uses CMock and Unity. Both Unity and CMock are important tools when it comes to testing of embedded systems. Ceedling is also a good tool for automation of building, but it is unknown if this can be integrated with Energy Micro's current system for building and testing.

Most of the problems that occurs when writing a test can be solved by using mock modules to remove dependencies, creating macros to redefine or change functionality that is not wanted or available during testing. The last thing that is used to change behavior during tests is to use conditional preprocessor expression. By using this we can for example use a pointer to a structure instead of a pointer to a fixed memory address. Not all can be solved by having good testing framework, it is also necessary to structure the code so that it is testable.

One complexity in testing for Energy Micro is that examples shall work with four different IDEs, examples shall work on different development boards and the peripheral API shall work for all EFM32 microcontrollers. The tests for the examples and peripheral APIs can be reused, by compiling and running the tests for each microcontroller, which is defined for the preprocessor. Due to for example different peripheral devices and sizes of memory for each microcontroller, it may be necessary to modify the tests in some cases. This part of testing has not been done in this study, and there is still work left, to find out how to solve this in an efficiently way.

The second research question for the thesis is "Show that CMock is a suitable framework for creating mock objects, which can simulate the behavior of the hardware, which is not available during testing on a native system." CMock is easy to configure and creates mock modules that works well during tests. There are some problems with structures and custom types and that inline methods are not accepted as input, but these problems are manageable. By having a framework that can generate mock modules and a unit test framework to do assertions, unit tests can be written to test software on a native system. When having tests that can cover most of the source code, errors in the system will be reduced. One of Energy Micro's goals is reduce the number of errors in the code before release. CMock and Unity should solve this, since adding tests will reduce the number of errors and testing with CMock and Unity will work on Energy Micros code.

After writing tests for the software library, the library will consist of a lot of test instrumentation code, which is only used to test the system. Before release of the code to customers, there will be necessary to remove all the test code instrumentation from the source code. Remove of test instrumentation can be done by a script, and the reason for removing the test instrumentation is that the customers have no need for this, and will it make the code harder to read and understand.

Advice 9: Most of the problems that occur during testing can be solved by using mock modules, macros and redefined functionality during test by using a preprocessor.

Advice 10: CMock works well in generation of mock modules that are needed for testing. By using a tool that automatically generates mock modules, developers can reduce time used for testing since they do not need to write the mock modules manually.

Advice 11: Before release of code, it will be necessary to remove test instrumentation code, to make the code more readable and easier to understand.

5.4 Test Strategy

This thesis has only looked at software testing and mainly at software unit. Section 2.3 gives a brief overview of both hardware and software testing. Since much of the code involves interaction and calls to lower layer in the software library, it is necessary to remove the dependencies during software unit testing. To be able to remove these dependencies, mock modules is an important part of the solution. By using these modules in tests, the unit can be tested in isolation, and the behavior can be tested automatically. Unit testing is important to remove errors in the unit that is tested. These tests can be used to verify changes during modification, and that refactoring do not change or break existing functionality. The unit tests will find and reduce the number of errors in the units, but will not test the interaction between software components. Unit testing is most important for the examples and drivers. It is also important for the peripheral API, even though this layer mainly consists of an abstraction layer that write configurations to memory registers, the layer consist of some logic that needs to be tested.

Software integration testing is testing of the interactions between software subsystems. When doing software integration testing, use of mock modules will be less important, since most assertions will be done by verifying that the correct registers are set by the software. Mock modules are needed when hardware interaction is necessary to be able to capture data that is written or inject data response when data is read.

Integration testing is testing to verify correct behavior when software interacts with hardware. Integration testing will normally require more manually work by the testers, since it often will require user interaction during testing. This is especially true when testing the examples, but testing of the peripheral APIs this can be done more or less automatic by unit tests. Testing on hardware is also the most correct place to test the hardware abstraction layer, but testing on a host computer can also reveal errors, and it will support test-driven development.

Energy Micro wants to reduce the number of errors before release of software. In addition they have many examples that need to be tested, so an additional goal is to be able to minimize the testing of examples on hardware, since this is time consuming. This means that Energy Micro needs to reduce integration testing of the examples. By having code that is covered by unit tests, where software is tested both with software unit tests and software integration tests, the number of errors may be reduced, because unexpected behavior will be found by the unit tests. When these errors are found early, the need for testing on hardware is reduced. Testing on hardware will still be necessary, but the testing can be focused on examples that are directly affected by the changes in the software.

Advice 12: All errors cannot be found by using unit test. Use of integration testing will still be necessary to verify that all parts of the system works as expected.

Advice 13: It is important to create a test strategy, so that the company has a plan for how testing of the software library shall be done.

5.5 Agile Development and Test-driven Development

One of the goals for Energy Micro when it comes to unit testing is to take advantage of agile development, with unit testing and test-driven development. The main changes needed to take advantage of agile development are related to how the project is structured and that development is done in iterations. By developing in iterations, it is possible to make decision late and adapt changes that occur during the development. Automated testing with unit tests is often used in agile development, and test-driven development is a technique that guarantees that tests are written for code that is developed. To use TDD, it will be necessary to have test frameworks that will support the test activities. Unity and CMock will work with test-driven development.

Automated building and testing tools is also important to support test-driven development, since tests are run often by the developers. Ceedling is a tool that is developed to support test-driven development with Unity and CMock. Ceedling was tested and has good features to perform testing during TDD. It can be difficult for Energy Micro to take advantage of using Ceedling since they already have an environment for compiling and testing their software. Another issue is that Energy Micro does not develop against a standalone product for a microcontroller, but creates many different examples for many different microcontrollers. How this support is in Ceedling is unknown.

Much of the code and examples that are written by Energy Micro is without tests, which makes it hard to do automatic regression testing and detect errors that are created during development and modification of existing functionality. Test-driven development is addressed to writing test before creating any code, but when changing code without tests, there will be a need to write tests for the unit that are going to be changed first. When tests are created it is possible to do modifications or do refactoring, since the test now will fail if some of the existing behavior is broken.

Section 2.2.4 refers to an article of how TDD can be done effectively for development on embedded systems. In section 2.4.3 we show how to apply test-driven development on existing code without tests.

In the experiment, half the participants had access to tests that was created before they started on a task. This is test-driven development, where tests are written before code is implemented, without the refactoring step. A problem that was observed is that it is difficult to understand the feedback that is given by tests that are failing. In the experiment all the tests was created first, and many of these tests covered almost all the functionality. By having smaller units the feedback that is given during the tests will be easier to understand.

Advice 14: Keep the size of the units under development in each of the iterations in TDD as small as possible.

Advice 15: Use of building tools that supports test-driven development will help the developers. Ceedling is a tool that can support test-driven development with Unity and CMock.

Advice 16: Before existing code without tests is changed, unit tests should be written.

5.6 Regression Testing

The last goal for Energy Micro is to be able to refactor or change the code on a later time, without breaking the code. Automatic regression testing with unit tests will be able to detect errors that are created, both in the software unit (by running the software unit tests), and errors that may occur in the in the interaction between software components. Not all errors can be found by the unit tests, so there will still be need for testing on hardware, to ensure that it behaves as expected. But by having good code coverage, the need for integration tests will be lower.

Good code coverage with tests is the most important part of being able to test changes when refactoring, since most of the functionality will be tested and asserted to see if it behave as before the changes was done. Even for a software library that do not have many tests, much can be solved by adding tests to the code before changes are done, so at least these changes are tested. But these changes can still break the functionality in components that depends on the unit that was changed. Errors can also be created if the model that the developer creates tests for do not match the real model. To detect errors like these, integration tests are needed.

Another way to reduce the need for testing all examples, is to be able to trace the changes that are done to the code. Doing a change does not necessarily mean that all examples will need to be tested. There is no need to test examples that do not use the component that was changed. By having a table or matrix that shows which examples uses which modules. It will be possible to see which examples that actually needs to be tested.

When testing during development or with test-driven development it is not possible to carry out all the tests every time. A repository that has a good coverage would consist of several thousand tests and even more assertions. To be able to verify that all tests are passing, it will be necessary to run the tests incrementally. There are two suggestions for this, one is to run night builds and run all tests that are available for the software, or run them on commits to the repository. The last can be difficult if there are a large amount of tests or commits to the repository, since the build servers not will be able to keep up.

Advice: 17 To reduce the need for integration testing, it will be useful to have a table or matrix showing the dependencies between examples and modules.

Advice: 18 A developer cannot run all tests every time. Therefore it will be necessary to run nightly builds that run all tests in the repository.

CHAPTER 6 CONCLUSION

Below follows the conclusion for the thesis and further work for testing of embedded software.

6.1 Conclusion

The first part of the thesis looks at testing of Energy Micro's software library, where examples, drivers and peripheral APIs are tested. The second part is an experiment to check that testing with mock modules will reduce the number of errors, compared to not using any techniques for automated testing. In the beginning of the project, two research questions were defined.

RQ1: Is it possible to reduce the number of errors during development of new features, modification and refactoring of existing functionality by using mock modules to verify the desired behavior of the system?

The experiment showed that testing with mock modules would reduce the number of errors that are created during implementation and modification of existing functionality.

RQ2: Show that CMock is a suitable framework to create mock objects, which can simulate the behavior of the hardware, which is not available during testing on a native system.

CMock is a suitable tool for automatic generation of mock modules from header-files. CMock do not accept inline methods or macro functions as input, but this can easily be fixed by creating a script that modify the header files. In addition, CMock do not have any real support for custom types and structure, but there are several ways that this can be solved.

By using Unity and CMock as frameworks for testing, Energy Micro will be able to reduce the number of errors in their software and make it possible to verify that functionality is not broken during modification and refactoring of software. By having the ability to automate testing of the software, Energy Micro can take advantage of test-driven development. Use of CMock to generate mock modules and use the mock modules to solve dependencies and perform testing without hardware, will reduce the amount of errors in embedded software.

6.2 Further Work

The most important further work will be to check the validity of the experiment by trying the experiment in a real situation, where professional developers use mock modules in the project and trying out test-driven development as a methodology. By using mock modules in a real environment, it can be tested whether the use of mock modules really works, and that the result in the experiment is significant.

Ceedling is a tool that makes testing easier with Unity and CMock, and which can support test-driven development. This thesis has looked only briefly on Ceedling and just made a simple test with it on Energy Micro software. It is necessary to make a more detailed study of this tool. In addition it will be necessary to look at how tests can be efficiently reused, when tests are run for several different microcontrollers that have different features and other differences.

This study has mainly looked at testing of software on a native computer, and both testing on emulators and development board is possible. It would be interesting to see if testing on an emulator or simulator could be more efficient than testing the software with mock modules. Testing the software on hardware is important, so looking for solution that can test the software automatic on hardware is an important issue. Looking for solution for how to reduce the necessary user interaction during testing on hardware is also an issue.

BIBLIOGRAPHY

[1] ARM Cortex-M. http://www.arm.com/products/processors/cortex-m/. [Online; accessed 21. February 2013].

[2] Ceedling Intro. http://throwtheswitch.org/white-papers/ceedling-intro.html. [Online; accessed 21. April 2013].

[3] CMock Intro. http://throwtheswitch.org/white-papers/cmock-intro.html. [Online; accessed 08. March 2013].

[4] CMock Summary. https://github.com/ThrowTheSwitch/CMock/blob/master/docs/CMock_Summary.md. [Online; accessed 21. May 2013].

[5] Energy Micro. http://www.energymicro.com/. [Online; accessed 21. February 2013].

[6] RAKE – Ruby Make. http://rake.rubyforge.org/. [Online; accessed 1. June 2013].

[7] Unity Intro. http://throwtheswitch.org/white-papers/unity-intro.html. [Online; accessed 08. March 2013].

[8] Unity Summary. https://github.com/ThrowTheSwitch/Unity/blob/master/docs/Unity%20Summary.txt. [Online; accessed 21. May 2013].

[9] White Papers - CMock, pointers, and structs. http://throwtheswitch.org/white-papers/cmock-pointers-and-structs.html. [Online; accessed 21. May 2013].

[10] White Papers - Mocking inline functions or macros. http://throwtheswitch.org/white-papers/mocking-inline-functions-or-macros.html. [Online; accessed 01. March 2013].

[11] Energy Micro AS. *EFM32GG Reference Manual*, 0.96 edition, 24 April 2012.

[12] Energy Micro AS. *User Manual Starter Kit EFM32GG-STK3700*, 0.11 edition, 31 May 2012.

[13] Bart Broekman and Edwin Notenboom. *Testing embedded software*. Addison-Wesley, 2003.

[14] M. Dowty. Test driven development of embedded systems using existing software test infrastructure. *University of Colorado at Boulder, Tech. Rep., Mar*, 2004.

[15] Michael Feathers. *Working effectively with legacy code*. Prentice Hall, 2004.

[16] J.W. Grenning. *Test Driven Development for Embedded C*. Pragmatic Bookshelf Series. Pragmatic Bookshelf, 2011.

[17] ISO/IEC JTC1 SC22 WG14. ISO/IEC 9899:TC2 Programming Languages - C. Technical report, May 2005.

[18] M.J. Karlesky, W.I. Bereza, and C.B. Erickson. Effective test driven development for embedded software. In *Electro/information Technology, 2006 IEEE International Conference on*, pages 382–387. IEEE, 2006.

[19] Kim H Pries and Jon M Quigley. *Testing complex and embedded systems*. CRC Press, 2011.

[20] S. Raffeiner. Functionality and design of the CMock framework.

[21] Mark VanderVoord. *Embedded Testing with Unity and CMock*. Throw-The-Switch Productions, 2010.

[22] M. Williams. Low pin-count debug interfaces for multi-device systems. 2009.

[23] C. Wohlin. *Experimentation in Software Engineering: An Introduction*. The Kluwer International Series in Software Engineering. Kluwer Academic, 2000.

[24] W. Wolf. *Computers as Components: Principles of Embedded Computing System Design*. The Morgan Kaufmann Series in Computer Architecture and Design. Elsevier Science, 2008.

LIST OF ABBREVIATIONS

ACMP	Analog Comparator
ADC	Analog to Digital Converter
AES	Advanced Encryption Standard
API	Application Programming Interface
CMSIS	Cortex Microcontroller Software Interface Standard
CMU	Clock Management Unit
EFM	Energy Friendly Microcontrollers
EFR	Energy Friendly Radios
emlib	Energy Micro Peripheral Library
GCC	The GNU Compiler Collection
GPIO	General Purpose Input/Output
HAL	Hardware Abstraction Layer
IDE	Integrated Development Environment
JTAG	Joint Test Action Group
LCD	Liquid Crystal Display
LED	Light-emitting Diode
PRS	Peripheral Reflex System
RISC	Reduced Instruction Set Computing
SWO	Serial Wire Output
TDD	Test-driven Development

UART	Universal Asynchronous Receiver/Transmitter
YAML	YAML Ain't Markup Language

APPENDIX A EXPERIMENT GUIDELINES

A.1 Introduction

In this experiment you will implement five small examples that use some of the features that are available on EFM32 Giant Gecko STK3700 starter kit. The code you implement will be tested later, and the results will be compared between the groups that have access to unit tests and the participants without tests during implementations.

The goal for the experiment is to see if regression testing with unit tests and mock object/modules can reduce the number of errors before software is uploaded to hardware.

In the experiment you will finish five simple tasks, which will demonstrate some of features on the development board. The examples that you implement will use the segment LCD, LEDs, buttons and the capacitive sensor on the development board. Each of the tasks have a set of requirements that must be implemented and instructions of what to do in each task.

After the implementation is done and the binary file is uploaded to the development board, you are not allowed to modify the code for the current task.

The participants will be divided in two groups. One on of the groups will have access to unit tests during the experiment, and can run these whenever they want to. The unit tests will use mock modules to "simulate" the behavior of the hardware, since hardware is not available during unit testing. The mock modules are used to verify that the correct function calls are performed, with correct parameters. Return values are injected from the mock modules. The expectations and injection of return values is set in the tests. When tests are run, the correct number of calls and parameters to the function calls are compared.

A.2 Instruction

This section consists of instructions for downloading the necessary code, overview of the documentation, instructions for upload of code to microcontroller over J-Link and a short introduction to the developpment board.

A.2.1 Code

Before you begin on the experiment you will need to download and unpack the necessary files. The tools and source code are packed. The archive includes the CodeSourcery compilers, energyAware Commander that is used for uploading binary to the microcontroller and the source code.

Download the source code and tools with:

```
# wget http://folk.ntnu.no/eribe/eutf/zebkrcdu.tar.gz
```

Unpack the files:

```
# tar zxvf zebkrcdu.tar.gz
```

All the necessary examples, the compiler and tools will now be available in `eutf/` folder.

A.2.2 Documentation

You should have received the necessary documentation for the experiment. The documentation includes the Doxygen documentation for the following modules:

- The GPIO (General Purpose Input/Output) peripheral API.
- Segment LCD driver.
- Capacitive sense driver.
- The Board Support Package for the LEDs on the starter kit.

The documentation can also be opened on the computer with:

```
# evince eutf/docs/refman.pdf
```

A.2.3 Upload to development board

Upload to development board will be done with energyAware Commander, this tool can be found in:

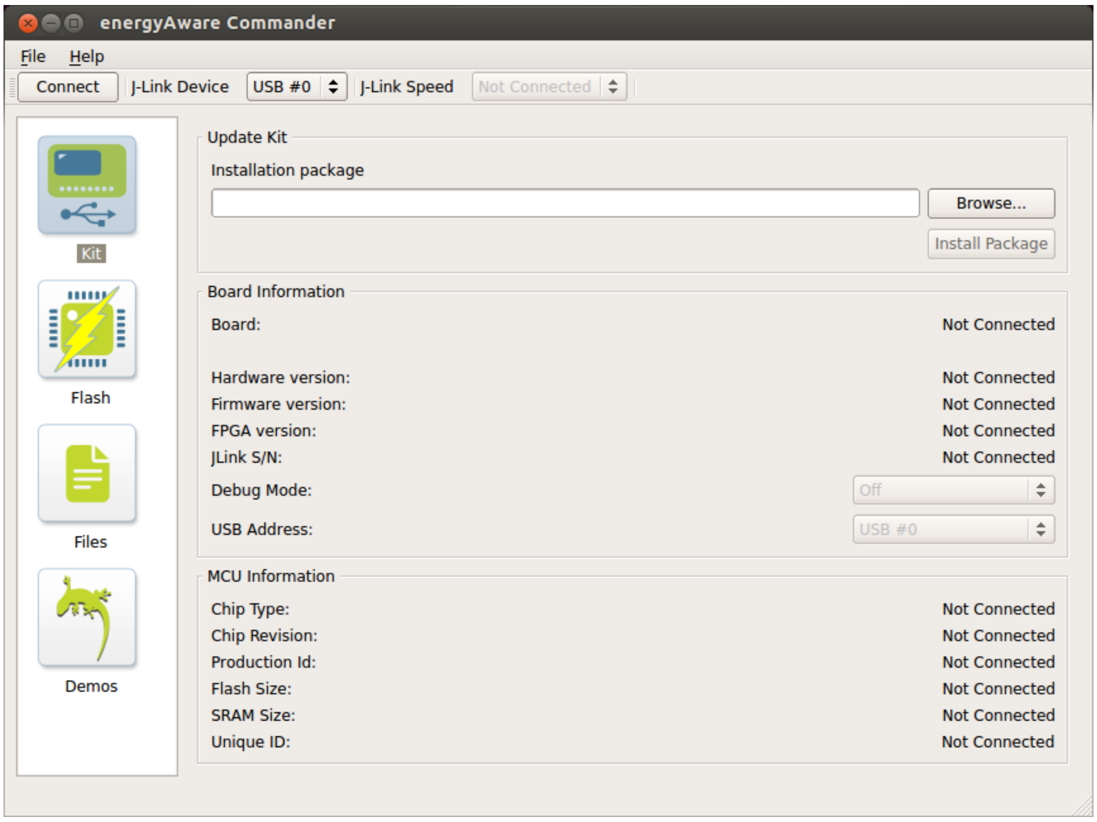
```
# cd  /eutf/tools/energymicro
```

The tool is started with:

```
# ./eACommander.sh
```

Figure A.1 shows the startup screen, to connect to the development chose the correct J-Link device and click 'connect'. To upload a binary to the board, click 'flash' on the left side.

After clicking "flash", browse the file you want to upload, and click 'Flash EFM32'. The binary should now be uploaded to the development board. An example is shown in Figure A.2



Update Kit

Installation package

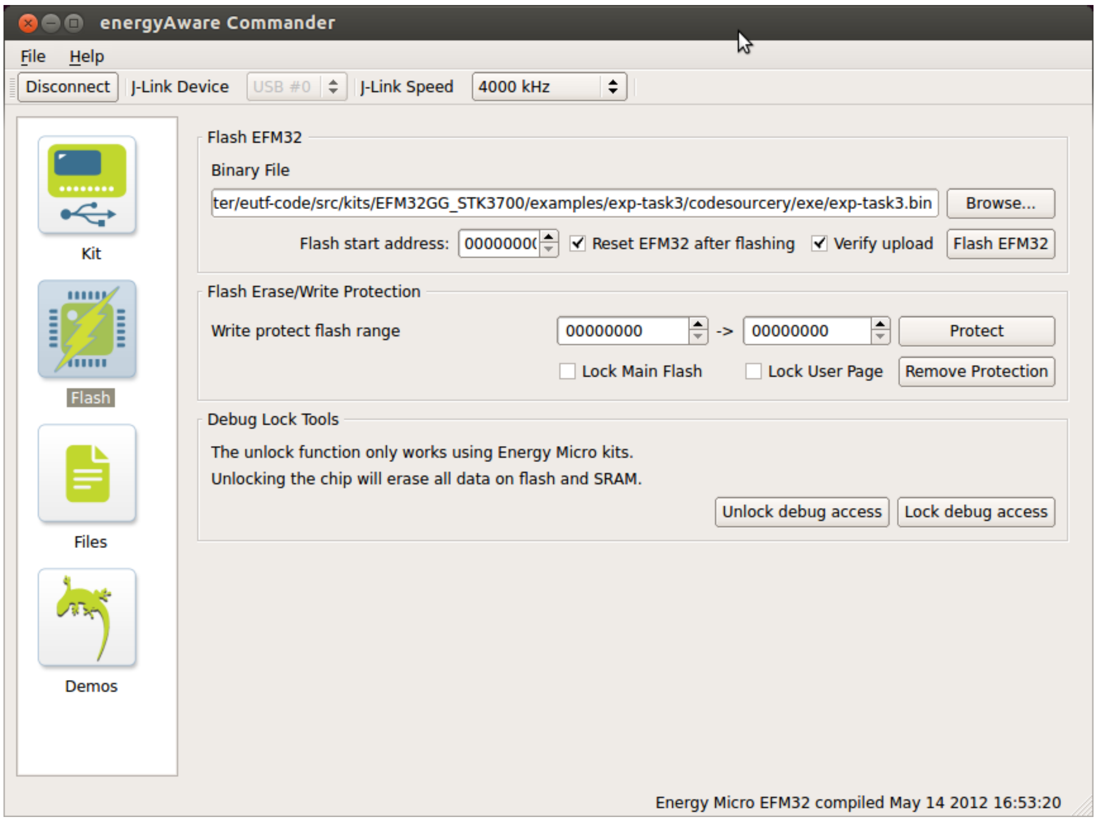
Board Information

Board:	Not Connected
Hardware version:	Not Connected
Firmware version:	Not Connected
FPGA version:	Not Connected
JLink S/N:	Not Connected
Debug Mode:	Off
USB Address:	USB #0

MCU Information

Chip Type:	Not Connected
Chip Revision:	Not Connected
Production Id:	Not Connected
Flash Size:	Not Connected
SRAM Size:	Not Connected
Unique ID:	Not Connected

Figure A.1: Connect with energyAware Commander



Flash EFM32

Binary File

ter/eutf-code/src/kits/EFM32GG_STK3700/examples/exp-task3/codesourcery/exe/exp-task3.bin

Flash start address: 0000000(☑ Reset EFM32 after flashing ☑ Verify upload

Flash Erase/Write Protection

Write protect flash range 00000000 -> 00000000

☐ Lock Main Flash ☐ Lock User Page

Debug Lock Tools

The unlock function only works using Energy Micro kits.
Unlocking the chip will erase all data on flash and SRAM.

Energy Micro EFM32 compiled May 14 2012 16:53:20

Figure A.2: Upload with energyAware Commander

A.2.4 Development board

An overview of the development board can be seen in Figure A.3, the figure is taken from the STK3700 user manual. In this experiment LEDs and the segment LCD will be used as output, and push-buttons and touch slider as input.

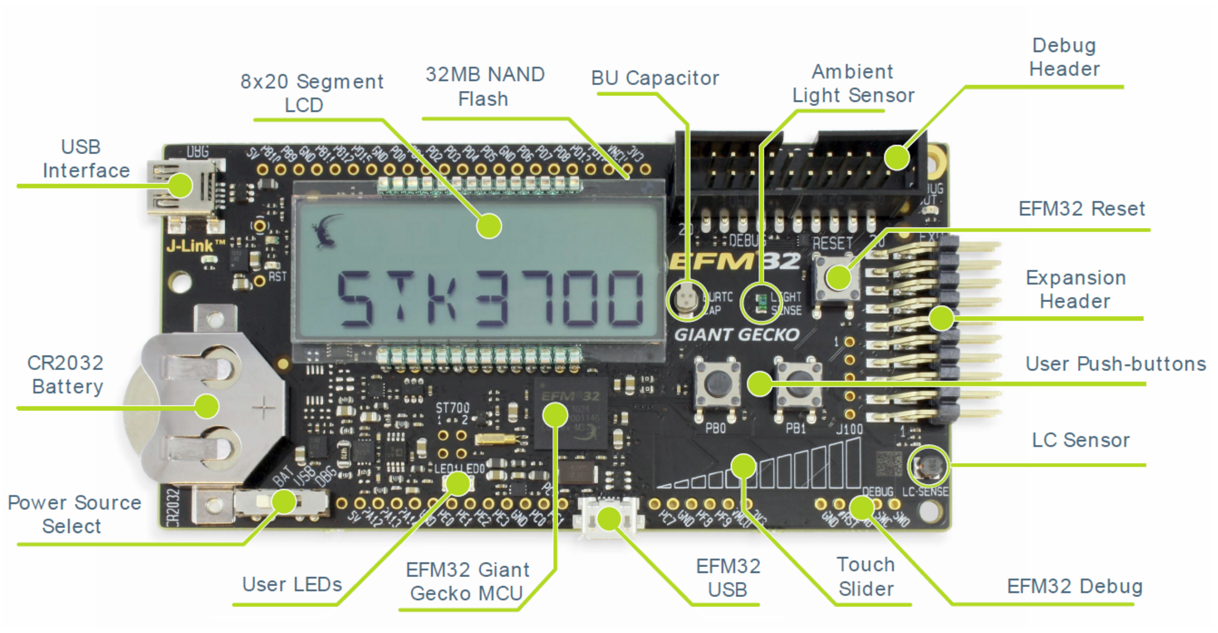


Figure A.3: STK3700 Hardware Layout [12]

Segment LCD

The driver documentation for the segment LCD is found on page 38 of the documentation that is handed out. The segments that will be used on the display are:

Text: It is possible to write 7 characters to the text segment on the display.

Numbers: 4 digits can be written to the number part of the segment display.

Ring indicators: The ring indicator consists of eight indicators that form a ring.

User LEDs

On the development board there is two LEDs, these will be used in most of the task. Documentation is found on page 27 of the Doxygen documentation. LED1 is the left LED on the development board, and LED0 is the right LED.

User push-buttons

There is two push-buttons on the board that will be used for input. The necessary documentation for this is found in the documentation that is handed out. The buttons will use the GPIO peripheral API.

- Button 0 (left button) is connected on pin 9, port B.
- Button 1 (right button) is connected on pin 10, port B.

Touch slider

The touch slider uses the capacitive sensor. It will be used for input in the last task, and documentation for this is found in the capacitive sense driver on page 30, of the Doxygen documentation provided.

A.3 Tasks

The experiment consists of five tasks, which follow below.

A.3.1 Task 1

This task will finish an application, which is a counter that starts on a given value and counts downwards. Your task will be to initialize the LEDs and the segment LCD. In addition you will have to implement the behavior when the counter reaches zero.

Requirements

Below are the requirements you will need to implement.

REQ1-1 The system shall initialize the segment LCD without voltage boost.

REQ1-2 The system shall initialize the LEDs on the development board.

REQ1-3 The system shall write "RESET" centered on the displayed, when the counter reach zero.

REQ1-4 The LEDs shall blink alternating, when the counter reach zero.

Instructions

The project folder for this task is:
`# eutf/src/kits/EFM32GG_STK3700/examples/exp-task1/`
The source file for the task is found in:
`# eutf/src/kits/EFM32GG_STK3700/examples/exp-task1/exp-task1.c`
The first part of the task is to initialize the segment LCD and the LEDs on the board, this must be implemented in the `main()` function. The documentation for the LCD is found on page 38, and the board support package LEDs is on page 27 in the Doxygen documentation. The segmented LCD shall be initialized without voltage boost.

The second part is to implement the behavior when the counter reaches zero, this will be implemented in `loop()`. The wanted behavior is that the LEDs on the board blink alternately and that "RESET" is displayed centered on the display.

After implementation you can run the tests. The tests are found in:

```
# eutf/src/kits/EFM32GG_STK3700/examples/exp-task1/tests
```

The tests are compiled and run with:

```
# make -f Makefile.exp-task1.test
```

The results of the unit tests are written to the screen and you can correct your code and then re-run the tests until you are finished and want to upload your code to the development board.

To compile the tests go to the folder below:

```
# eutf/src/kits/EFM32GG_STK3700/examples/exp-task1/codesourcery
```

and run the following command:

```
# make -f Makefile.blink
```

If you have a problem with compiling your code, ask for help.

Upload the binary file to the development board by using the upload instructions. The binary file is found in:

```
# eutf/src/kits/EFM32GG_STK3700/examples/exp-task1/codesourcery/exe/blink.bin
```

Results

If you have any comment, please write them in the box below:

A.3.2 Task 2

The task is to implement the functionality needed to be able to turn on and off LEDs by using the push buttons on the development board. In addition the segmented LCD will be used to display the total count of clicks on each push button.

Requirements

The requirements for this task follow below.

REQ2-1 Buttons shall be initialized and set up for interrupts.

REQ2-2 The LEDs shall toggle on click, the left LED is toggled on left clicks and the right LED is toggled on right clicks.

REQ2-3 The button shall give an interrupt on falling-edge.

REQ2-4 An interrupt shall be cleared after an interrupt.

REQ2-5 The segmented LCD shall display "[left-clicks]/[right-clicks]".

Description

The project folder for this task is:
`# eutf/src/kits/EFM32GG_STK3700/examples/exp-task2/`
The source file for the task is found in:
`# eutf/src/kits/EFM32GG_STK3700/examples/exp-task2/exp-task2.c`
The first part is to configure the buttons as input in the GPIO (General Purpose Input/Output), which will be done in `buttons_setup()`. Button 0 is connected to pin 9 on port B and button 1 is connected to pin 10 on port B, the function to setup this is `GPIO_PinModeSet()` and can be found on page 16 in the documentation. Description of `GPIO_Mode_TypeDef` and `GPIO_Port_TypeDef` is found on page 10 and 11. It will also be necessary to configure GPIO interrupts, which is done with `GPIO_IntConfig()`, and the interrupt should occur on falling edge, see documentation on page 14.

The second part is to clear a pending interrupt, this is done with `GPIO_IntClear()`, and documentation is available on page 13. Clearing of interrupts is done in the interrupt handlers `GPIO_ODD_IRQHandler()` and `GPIO_EVEN_IRQHandler()`. `GPIO_IntClear()` takes *flags* as input, which means you will have to set the correct pin to clear. The bit string '000...01000' is the flag to clear the interrupt for pin 3. Use either an integer value or use the left shift operator.

The last part is to make the LEDs turn on/off on clicks and write the total number of clicks to the segment LCD on an interrupt. Implement this where you want to make it work, but do not modify `loop()`.

After implementation you can now run the tests. The tests are found in:
`# eutf/src/kits/EFM32GG_STK3700/examples/exp-task2/tests`
The tests are compiled and run with:
`# make -f Makefile.exp-task2.test`
The results of the unit tests are written to the screen, and can correct your code and then re-run the tests until you are finished and want to upload your code to the development board.

To compile the tests go to the folder below:
`# eutf/src/kits/EFM32GG_STK3700/examples/exp-task2/codesourcery`
And run the following command:
`# make -f Makefile.blink`
If you have a problem with compiling your code, ask for help.

Upload the binary file to the development board by using the upload instructions. The binary file is found in:
`# eutf/src/kits/EFM32GG_STK3700/examples/exp-task2/codesourcery/exe/blink.bin`

Results

If you have any comment, please write them in the box below:

A.3.3 Task 3

This task uses the buttons to perform simple binary operations on a binary bit string. Your task is to implement the behavior when each button is clicked and output the values of the bit string as specified in the requirements.

Requirements

Below are the requirements you will need to implement in this task.

REQ3-1 The binary bit string shall be inverted when the left push button is clicked.

REQ3-2 The binary bit string shall be incremented by one when the right push button is clicked.

REQ3-3 The number segment on the segment LCD shall show the decimal value of the binary bit string.

REQ3-4 The text segment shall display the seven last bits in the binary bit string.

Description

The project folder for this task is:
`# eutf/src/kits/EFM32GG_STK3700/examples/exp-task3/`
The source file for the task is found in:
`# eutf/src/kits/EFM32GG_STK3700/examples/exp-task3/exp-task3.c`
The first part is to implement the logic when the buttons is clicked. On left click, the bit string shall be inverted, on right click the bit string shall be incremented by one. The bit string is stored in an 8 bit unsigned integer, defined as *value*. The maximum value of this integer is 255. This functionality must be implemented in the interrupts handlers.

In the second part you should write the value of the integer to the number segment on the LCD. Documentation of the segment LCD is found on page 38. Implement this in `loop()`.

The last part is to write the 7 last bits of the value to the text segment on the LCD. Implement this in `loop()`.

After implementation you can now run the tests. The tests are found in:

```
# eutf/src/kits/EFM32GG_STK3700/examples/exp-task3/tests
```

The tests are compiled and run with:

```
# make -f Makefile.exp-task3.test
```

The results of the unit tests are written to the screen and you can correct your code and then re-run the tests until you are finished and want to upload your code to the development board.

To compile the tests go to the folder below:

```
# eutf/src/kits/EFM32GG_STK3700/examples/exp-task3/codesourcery
```

And run the following command:

```
# make -f Makefile.blink
```

If you have a problem with compiling your code, ask for help.

Upload the binary file to the development board using the upload instructions. The binary file is found in:

```
# eutf/src/kits/EFM32GG_STK3700/examples/exp-task3/codesourcery/exe/blink.bin
```

Results

If you have any comment, please write them in the box below:

A.3.4 Task 4

This task is to modify the behavior when displaying a long string, since the segmented LCD only have 7 segments for text, and the string will have to loop over the display.

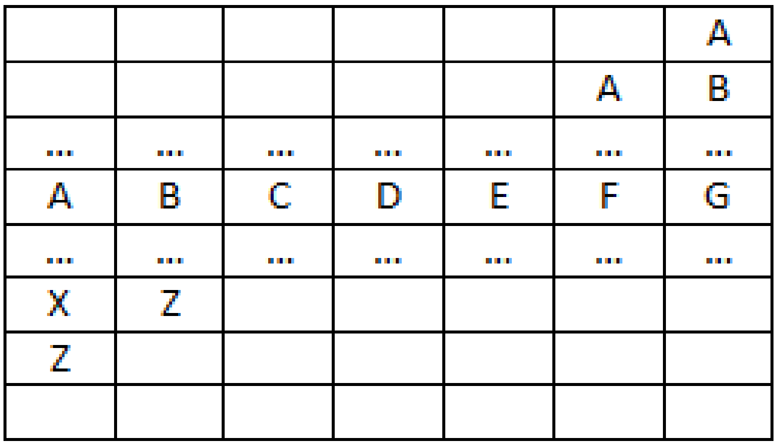
Requirements

Below are the requirements you will need to implement in this task.

REQ4-1 The string shall be written to the segment LCD.

REQ4-2 The display shall loop through the string as shown in Figure A.4.

REQ4-3 The loop shall continue from A, when the end is reached.



						A
					A	B
...	...	...	...	...	...	...
A	B	C	D	E	F	G
...	...	...	...	...	...	...
X	Z					
Z						

Figure A.4: Text Pattern

Description

The project folder for this task is:
`# eutf/src/kits/EFM32GG_STK3700/examples/exp-task4/`
The source file for the task is found in:
`# eutf/src/kits/EFM32GG_STK3700/examples/exp-task4/exp-task4.c`
The task is to make a long string rotate on the segment LCD. The string is defined as a char array defined as *alphabet*. The pattern that shall be displayed can be seen in Figure A.4. Implement this in `loop()`. You can use `snprintf(char, size_t, const char, ...)` to temporary write the output to the string defined as *value*.

`snprintf(char *str, size_t size, const char *format, ...)`

***str** The character string to be written.

size The number of bytes to write to *str

***format** The format string

... Variable arguments, defined by the format string

`snprintf(output, 8, "%s", &alphabet[i])` will write a 7 byte string from the i'th byte of the char array (character string) to the array *output*, the last char in output will be '\0'.

After implementation you can run the tests. The tests are found in:
`# eutf/src/kits/EFM32GG_STK3700/examples/exp-task4/tests`
The tests are compiled and run with:
`# make -f Makefile.exp-task4.test`
The results of the unit tests are written to the screen and you can correct your code and then re-run the tests until you are finished and want to upload your code to the development board.

To compile the tests go to the folder below:
```
# eutf/src/kits/EFM32GG_STK3700/examples/exp-task4/codesourcery
```
And run the following command:
```
# make -f Makefile.blink
```
If you have a problem with compiling your code, ask for help.

Upload the binary file to the development board by using the upload instructions. The binary file is found in:
```
# eutf/src/kits/EFM32GG_STK3700/examples/exp-task4/codesourcery/exe/blink.bin
```

Results

If you have any comment, please write them in the box below:

A.3.5 Task 5

In this task the segment LCD is used as output and capacitive touch slider is used as input.

Requirements

The requirements for the last task in the experiment:

REQ5-1 The system shall initialize and setup the touch slider.

REQ5-2 The indicator ring on the segment display shall show the position of the touch slider.

REQ5-3 The number segment on the segment LCD shall display the decimal value of the touch slider position.

REQ5-4 The segment LCD shall be blank when there is no input on the touch slider.

Description

The project folder for this task is:
```
# eutf/src/kits/EFM32GG_STK3700/examples/exp-task5/
```

The source file for the task is found in:

```
# eutf/src/kits/EFM32GG_STK3700/examples/exp-task5/exp-task5.c
```

The first part is to initialize and set up the capacitive sense system. Do not set the system in sleep. This is important so it is possible to poll for values from the touch slider. The documentation for capacitive sense driver is found on page 30. Implement the initialization and setup in `main()`.

The next part is to finish `update_segment_ring(int)`. The segment ring consists of 8 segments, that need to be updated corresponding to the integer value parameter. The maximum value of this parameter is 48.

The last part is to update the segment number and segment ring on the display for each iteration. No number shall be displayed when no position is given from the slider. Do this in `loop()`.

After implementation you can now run the tests. The tests are found in:

```
# eutf/src/kits/EFM32GG_STK3700/examples/exp-task5/tests
```

The tests are compiled and run with:

```
# make -f Makefile.exp-task5.test
```

The results of the unit tests are written to the screen and you can correct your code and then re-run the tests until you are finished and want to upload your code to the development board.

To compile the tests go to the folder below:

```
# eutf/src/kits/EFM32GG_STK3700/examples/exp-task5/codesourcery
```

And run the following command:

```
# make -f Makefile.blink
```

If you have a problem with compiling your code, ask for help.

Upload the binary file to the development board by using the upload instructions. The binary file is found in:

```
# eutf/src/kits/EFM32GG_STK3700/examples/exp-task5/codesourcery/exe/blink.bin
```

Results

If you have any comment, please write them in the box below:

A.4 When Finished

When you are finished with all the tasks please deliver your code and this document.

Thanks for participating in this experiment!

If you have any comments please write them in the box on the next page.

APPENDIX B EXPERIMENT TEST CASES

B.1 Task 1

Table B.1 - Table B.5 shows the test cases that was used to test the expected behavior for the code that was implemented by the experiment participants.

Table B.1: Test case TC1-1

Header	Description
Description	Test that system behave as expected during countdown.
Tester	Experiment participant
Prerequisites	Implemented the functionality, compiled the source code and then uploaded the binary to the development board.
Feature	Counter is decremented and the "ring" on the segmented LCD is showing the progress.
Execution	1. Reset the microcontroller by clicking the reset button. 2. Look at the progress on the segmented LCD.
Expected result	2. Verify that number in the upper-left corner is counting down. 2. Verify that the "ring" on the segmented LCD grows for each eighth that is counted down.

Table B.2: Test case TC1-2

Header	Description
Description	Segment LCD initalized
Tester	Experiment participant
Prerequisites	Implemented the functionality, compiled the source code and then uploaded the binary to the development board.
Feature	The segment LCD is initialized correctly.
Execution	1. Reset the microcontroller by clicking the reset button. 2. Check that the segmented LCD is working.
Expected result	2. Verify that number in the upper-left corner is counting down.

Table B.3: Test case TC1-3

Header	Description
Description	BSP LEDs initialized.
Tester	Experiment participant
Prerequisites	Implemented the functionality, compiled the source code and then uploaded the binary to the development board.
Feature	Initialized LEDs is done.
Execution	1. Reset the microcontroller by clicking the reset button. 2. Wait for the counter to reach zero.
Expected result	2. When counter reach zero, the LEDs shall blink.

Table B.4: Test case TC1-4

Header	Description
Description	LEDs blink alternating.
Tester	Experiment participant
Prerequisites	Implemented the functionality, compiled the source code and then uploaded the binary to the development board.
Feature	LEDs shall blink alternating, when countdown is finished.
Execution	1. Reset the microcontroller by clicking the reset button. 2. Wait for the counter to reach zero.
Expected result	2. LEDs shall be blinking alternative.

Table B.5: Test case TC1-5

Header	Description
Description	RESET Centralized
Tester	Experiment participant
Prerequisites	Implemented the functionality, compiled the source code and then uploaded the binary to the development board.
Feature	Segment LCD shall display "RESET" when countdown is finished.
Execution	1. Reset the microcontroller by clicking the reset button. 2. Wait for the counter to reach zero.
Expected result	2. Segmented LCD should display "RESET". First and last text segment shall be empty.

B.2 Task 2

To verify correct behvior of the implemented code in task 2, test cases Table B.6 - Table B.11was executed.

Table B.6: Test case TC2-1

Header	Description
Description	Existing functionality is working.
Tester	Experiment participant
Prerequisites	Implemented the functionality, compiled the source code and then uploaded the binary to the development board.
Feature	Check that CHIP is initalized and loop is running infinitely.
Execution	1. Reset the microcontroller by clicking the reset button. 2. Click the buttons a few times.
Expected result	2. Verify that left LED is turned on or LCD is updated on each click.

Table B.7: Test case TC2-2

Header	Description
Description	Buttons is working.
Tester	Experiment participant
Prerequisites	Implemented the functionality, compiled the source code and then uploaded the binary to the development board.
Feature	Buttons shall be initialzed and interrupts configured correctly.
Execution	1. Reset the microcontroller by clicking the reset button. 2. Click the left button. 3. Click the right button.
Expected result	2. Verify the segmented LCD shows "1/0". 3. Verify that LEDs or LCD is updated.

Table B.8: Test case TC2-3

Header	Description
Description	LEDs updated on click.
Tester	Experiment participant
Prerequisites	Implemented the functionality, compiled the source code and then uploaded the binary to the development board.
Feature	LEDs shall be toggle when a button is clicked.
Execution	1. Reset the microcontroller by clicking the reset button. 2. Click the left button. 3. Click the right button. 4. Click the left button.
Expected result	2. Verify that left LED is turned on. 3. Verify that right LED is turned on. 4. Verify that left LED is turned off.

Table B.9: Test case TC2-4

Header	Description
Description	Interrupts is cleared.
Tester	Experiment participant
Prerequisites	Implemented the functionality, compiled the source code and then uploaded the binary to the development board.
Feature	Interrupt shall be cleared when a interrupt occur.
Execution	1. Reset the microcontroller by clicking the reset button. 2. Click the left button. 3. Click the right button.
Expected result	2. Left LED shall only change state once and segment LCD is only incremented by one. 3. Right LED shall only change state once and segment LCD is only incremented by one.

Table B.10: Test case TC2-5

Header	Description
Description	Interrupt on falling edge.
Tester	Experiment participant
Prerequisites	Implemented the functionality, compiled the source code and then uploaded the binary to the development board.
Feature	The button shall create interrupt on falling edge.
Execution	1. Reset the microcontroller by clicking the reset button. 2. Click and hold the left button. 3. Releas left button. 4. Click and hold the right button. 5. Releas right button
Expected result	2. Verify that left LED is turned on and left counter is incremented to 1. 3. Left LEDs and segment LCD shall not change. 4. Verify that right LED is turned on and right counter is incremented to 1. 5. Right LEDs and segment LCD shall not change.

Table B.11: Test case TC2-6

Header	Description
Description	Display updated on click.
Tester	Experiment participant
Prerequisites	Implemented the functionality, compiled the source code and then uploaded the binary to the development board.
Feature	The number of clicks shall be displayed on the segment LCD.
Execution	1. Reset the microcontroller by clicking the reset button. 2. Click the left button. 3. Click the right button. 4. Click the left button.
Expected result	2. The text segment on the LCD shal display 1-0. 3. The text segment on the LCD shal display 1-1. 4. The text segment on the LCD shal display 2-1.

B.3 Task 3

Test cases for task 3 is found in Table B.12 - Table B.12.

Table B.12: Test case TC3-1

Header	Description
Description	Existing functionality is working.
Tester	Experiment participant
Prerequisites	Implemented the functionality, compiled the source code and then uploaded the binary to the development board.
Feature	Check that CHIP is initalized and loop is running infinitely.
Execution	1. Reset the microcontroller by clicking the reset button. 2. Click the buttons a few times.
Expected result	2. Verify that LCD is updated on each click.

Table B.13: Test case TC3-2

Header	Description
Description	Value incremented on right click.
Tester	Experiment participant
Prerequisites	Implemented the functionality, compiled the source code and then uploaded the binary to the development board.
Feature	The binary bit string shall be incremented by one when the right push button is clicked.
Execution	1. Reset the microcontroller by clicking the reset button. 2. Click the right button. 3. Click the right button. 4. Click the right button.
Expected result	2. The number segment display: 1 3. The number segment display: 2 4. The number segment display: 3

Table B.14: Test case TC3-3

Header	Description
Description	Value inverted on left click.
Tester	Experiment participant
Prerequisites	Implemented the functionality, compiled the source code and then uploaded the binary to the development board.
Feature	The binary bit string shall be inverted when the left push button is clicked.
Execution	1. Reset the microcontroller by clicking the reset button. 2. Click the left button. 3. Click the left button.
Expected result	2. The number segment display: 255 3. The text segment display: 0

Table B.15: Test case TC3-4

Header	Description
Description	Number is correctly updated on click.
Tester	Experiment participant
Prerequisites	Implemented the functionality, compiled the source code and then uploaded the binary to the development board.
Feature	The number segment on the segment LCD shall show the decimal value of the binary bit string.
Execution	1. Reset the microcontroller by clicking the reset button. 2. Click the right button. 3. Click the right button. 4. Click the left button. 5. Click the right button.
Expected result	2. The number segment display: 1 3. The number segment display: 2 4. The number segment display: 253 5. The number segment display: 254

Table B.16: Test case TC3-5

Header	Description
Description	Test that the functionality is implemented correcyly, and the display shows correct values.
Tester	Experiment participant
Prerequisites	Implemented the functionality, compiled the source code and then uploaded the binary to the development board.
Feature	The text segment shall display the seven last bits in the binary bit string.
Execution	1. Reset the microcontroller by clicking the reset button. 2. Click the right button. 3. Click the left button. 4. Click the right button. 5. Click the left button.
Expected result	2. The text segment displays "0000001". 3. The text segment display: "1111110" 4. The text segment display: "1111111" 5. The text segment display: "0000000"

B.4 Task 4

Test cases to verify expected behavior is found in Table B.17 - Table B.20

Table B.17: Test case TC4-1

Header	Description
Description	Existing functionality is working.
Tester	Experiment participant
Prerequisites	Implemented the functionality, compiled the source code and then uploaded the binary to the development board.
Feature	Check that CHIP is initalized and loop is running infinitely.
Execution	1. Reset the microcontroller by clicking the reset button. 2. Wait a few seconds.
Expected result	2. Verify that LCD is updated on each click.

Table B.18: Test case TC4-2

Header	Description
Description	Value is written to segment LCD.
Tester	Experiment participant
Prerequisites	Implemented the functionality, compiled the source code and then uploaded the binary to the development board.
Feature	The string shall be written to the segment LCD.
Execution	1. Reset the microcontroller by clicking the reset button. 2. Wait for the program to start.
Expected result	2. Verify that text is written to the display.

Table B.19: Test case TC4-3

Header	Description
Description	The pattern given in the requirements is correct.
Tester	Experiment participant
Prerequisites	Implemented the functionality, compiled the source code and then uploaded the binary to the development board.
Feature	The display shall loop through the string.
Execution	1. Reset the microcontroller by clicking the reset button. 2. Wait for the program to loop through the string.
Expected result	2. Verify that string follows the correct pattern.

Table B.20: Test case TC4-4

Header	Description
Description	Rotation restarts when reach end.
Tester	Experiment participant
Prerequisites	Implemented the functionality, compiled the source code and then uploaded the binary to the development board.
Feature	The loop shall continue from A, when the end is reached.
Execution	1. Reset the microcontroller by clicking the reset button. 2. Wait for the string to reach the end.
Expected result	2. When the end is reach, verify at the string starts from the beginning again.

B.5 Task 5

The test cases that was used for task 5 in the experiment is Table B.21 to Table B.25.

Table B.21: Test case TC5-1

Header	Description
Description	Existing functionality is working.
Tester	Experiment participant
Prerequisites	Implemented the functionality, compiled the source code and then uploaded the binary to the development board.
Feature	Check that CHIP is initalized and loop is running infinitely.
Execution	1. Reset the microcontroller by clicking the reset button. 2. Use the touch slider.
Expected result	2. Segment LCD shall change when finger is moved on the touch slider.

Table B.22: Test case TC5-2

Header	Description
Description	Caplesense is initialized.
Tester	Experiment participant
Prerequisites	Implemented the functionality, compiled the source code and then uploaded the binary to the development board.
Feature	The system shall initialize and setup the touch slider.
Execution	1. Reset the microcontroller by clicking the reset button. 2. Use the touch slider.
Expected result	2. Data should be read from the touch slider and be displayed on the segment LCD.

Table B.23: Test case TC5-3

Header	Description
Description	Segment ring correct.
Tester	Experiment participant
Prerequisites	Implemented the functionality, compiled the source code and then uploaded the binary to the development board.
Feature	The ring segment shall display the value from the touch slider.
Execution	1. Reset the microcontroller by clicking the reset button. 2. Set finger at the left side of the touch slider. 3. Move finger slowly to the right side of the touch slider. 4. Move finger slowly back to the left side of the touch slider.
Expected result	2. None of the segment rings is disabled. 3. The segment rings slowly increase, until a ring is formed. 4. The segment rings slowly decrease, until none ring indicators is visible.

Table B.24: Test case TC5-4

Header	Description
Description	Segment number correct.
Tester	Experiment participant
Prerequisites	Implemented the functionality, compiled the source code and then uploaded the binary to the development board.
Feature	Number segment shall display the value from the touch slider.
Execution	1. Reset the microcontroller by clicking the reset button. 2. Set finger at the left side of the touch slider. 3. Move finger slowly to the right side of the touch slider. 4. Move finger slowly back to the left side of the touch slider.
Expected result	2. The number segment shall display about 1. 3. The number slowly increases to 48. 4. The number slowly decreases to 1.

Table B.25: Test case TC5-5

Header	Description
Description	Display empty on no input.
Tester	Experiment participant
Prerequisites	Implemented the functionality, compiled the source code and then uploaded the binary to the development board.
Feature	The segment LCD shall be blank when there is no input on the touch slider.
Execution	1. Reset the microcontroller by clicking the reset button. 2. Set a finger on the touch slider. 3. Remove finger from the touch slider.
Expected result	2. Some values shall be displayed at the segment LCD. 3. No text, numbers or ring indicators is visible on the segment LCD.

APPENDIX C MOCK PREPERATION SCRIPT

This appendix includes the script that is used to prepare the header-files for mock generation with CMock. The script adds test instrumentation to inline static functions and includes directives in the header-file, preprocess the file, and start generation of the mock objects.

C.1 Script

The script to prepare header-files for mock generation is shown in Listing C.1.

```
require 'yaml'

@errors = 0

@config = nil

def parse_braces(line)
  chars = line.each_char.to_a
  count = Hash.new(0)
  chars.each { |ch| count[ch] += 1 }

  return (count["{"] - count["}"])
end

def include_accepted?(line)
  @config["acceptincludes"].each do |i|
    if line == "#include \"#{i}\""
      return true
    end
  end
  return false
end

def process_file(infile, outfile)
  curinfile = File.open(infile)
  curoutfile = File.new(outfile, "w")

  static_inline_line = 0
  static_parsing = false
  num_brace = 0

  curinfile.each_line do |line|
    line = line.rstrip
    curline = line.encode('UTF-8', 'binary', :invalid => :replace, :undef
        => :replace).split(' ')
    firstword = curline.shift
    if static_parsing
      num_brace += parse_braces(line)

      if num_brace != 0
        curoutfile.puts(line)
      else
        curoutfile.puts(line)
        curoutfile.puts("#endif")
        static_parsing = false
      end
    elsif include_accepted?(line)
      curoutfile.puts(line)
    elsif "#include" == firstword
      curoutfile.puts("#ifndef IGNOREINCLUDE")
      curoutfile.puts(line)
      curoutfile.puts("#endif")
    elsif "__STATIC_INLINE" == firstword
      static_inline_line = curinfile.lineno
      curline
      curoutfile.puts("#ifdef TEST")
      curoutfile.puts("#{curline.join(' ')};")
      curoutfile.puts("#else")

      if line.each_char.to_a.last == ";"
        curoutfile.puts(line)
        curoutfile.puts("#endif")
      else
        static_parsing = true
        curoutfile.puts(line)
      end
    else
      curoutfile.puts(line)
    end
  end

  if static_parsing
    puts "Err: STATIC INLINE was never finished, line: #{static_inline_line
        }"
    @errors = @errors + 1
  end

  curoutfile.close
  puts "Finished with #{infile}"
end

def directory_exist(dir)
  unless Dir.exist?(dir)
    puts "Directory '#{dir}' do not exist, create it and try again."
    exit
  end
end

def file_exist(file)
  unless File.exist?(file)
    puts "File '#{file}' do not exist, create it and try again."
    exit
  end
end

def get_include_files()
  out = ""

  @config["includepath"].each do |inc|
    out += "-I#{inc} "
  end

  return out
end

def get_defined_identifiers()
  out = ""

  @config["defineidentifiers"].each do |id|
    out += "-D#{id} "
  end

  return out
end

def main
  if ARGV.length != 1
    puts "Yaml config must be specified\nruby mockfix.rb <yaml-file>\n"
    puts "Includefiles must be put in inc/\nAll files that need to be
       preprocessed should go in input folder\n"
    exit
  end

  @config = YAML.load_file("#{ARGV[0]}")
  directory_exist(@config["inputpath"])
  directory_exist(@config["headeroutputpath"])
  directory_exist(@config["mockincludepath"])
  directory_exist(@config["mocksourcepath"])
  directory_exist(@config["tmppath"])
  directory_exist("#{@config["cmockpath"]}/mocks")
  file_exist(@config["cmockyaml"])
  file_exist(@config["cmockpath"])

  Dir.foreach(@config["inputpath"]) do |x|
    unless x == "." or x == ".."
      puts "Starting on #{@config["inputpath"]}/#{x}"
      process_file("#{@config["inputpath"]}/#{x}", "#{@config["
          headeroutputpath"]}/#{x}")
    end
  end

  if @errors > 0
    puts "Mock fixing finished with #{@errors} errors"
  else
    puts "Mock fixing finished without errors\nStarting preprocessing of
       headers and genereation of mock objects"
  end

  Dir.foreach(@config["headeroutputpath"]) do |x|

    unless x == "." or x == ".."
      unless system("gcc -E -P #{get_defined_identifiers()} #{
         get_include_files()} #{@config["headeroutputpath"]}/#{x} > #{
         @config["tmppath"]}/#{x}")
        puts "Error during preprocessing of #{x}"
      end
      unless system("ruby #{@config["cmockpath"]}cmock.rb -o#{@config["
         cmockyaml"]} #{@config["tmppath"]}/#{x}")
        puts "Error during generation of mocks"
      end
    end
  end

  unless system("cp #{@config["cmockpath"]}mocks/*.c #{@config["
     mocksourcepath"]}")
    puts "Error copying source-files to mock source path"
  end

  unless system("cp #{@config["cmockpath"]}mocks/*.h #{@config["
     mockincludepath"]}")
    puts "Error copying header-files to mock include path"
  end
end

main
```

Listing C.1: Preperation of Header-files Script

C.2 Config-file

Listing C.2 shows a sample YAML config file for the script to prepare header files.

```
#Path to yaml-config file for CMock
cmockyaml: MyYaml.yml
#Path to CMock library
cmockpath: ./
#Path to store temporary files
tmppath: ./tmp
#Include paths
includepath:
  - inc
#Path to header-files that are input to mock generation
inputpath: ./input
#Path to where prepared header-files shold be saved
headeroutputpath: ../../../src/emlib/incmock
#Path to output for mock module header- and source-files
mockincludepath: ../../../mocks/inc
mocksourcepath: ../../../mocks/src
#Accepted include directives, will not be removed during preprocessing.
```

```
acceptincludes:
  - efm32.h
  - em_device.h
  - em_part.h
#Files that need preprocessing before they are included. Functionality not
    implemented.
preprocessincludes:
  - efm32gg990f1024.h
#Identifiers for preprocessing
defineidentifiers:
  - TEST
  - IGNOREINCLUDE
  - EFM32GG990F1024
```

Listing C.2: Config-file for Mock Generation