FACULTY OF INFORMATION TECHNOLOGY AND ELECTRICAL ENGINEERING

Lari-Matias Orjala

UNIT TESTING METHODS FOR INTERNET OF THINGS MBED OS OPERATING SYSTEM

Master's Thesis
Degree Programme in Computer Science and Engineering
December 2019

Orjala L. (2019) Unit testing methods for Internet of Things Mbed OS operating system. University of Oulu, Degree Programme in Computer Science and Engineering, 40 p.

ABSTRACT

Embedded operating systems for Internet of Things are responsible for managing hardware and software in these systems. From the vast number of IoT operating system projects available, some projects are backed by large companies or institutes and some are developed completely by the open source community. IoT operating system testing focuses on the key features of IoT such as networking and limited resources.

In this thesis, problems in Mbed OS operating system testing methods are identified and a unit testing solution is implemented. The implemented unit testing framework allows developers to write and run unit tests. The framework is also integrated into Mbed OS continuous integration to increase test coverage.

This thesis shows how functional testing and unit testing are the most common types of testing in open source embedded operating system projects. Mbed OS unit testing framework results shows how running tests on PC platforms is faster than running tests on IoT devices. This framework also enables developers to write unit tests more freely and improve Mbed OS development process.

The implemented unit testing framework solved issues in Mbed OS testing but more in depth research is needed to improve testing methods further.

Keywords: IoT, Open source, Unit testing, Functional testing, Regression testing, Continuous integration

Orjala L. (2019) Yksikkötestausmenetelmät esineiden internet Mbed OS käyttöjärjestelmälle. Oulun yliopisto, Tietotekniikan tutkinto-ohjelma, 40 s.

TIIVISTELMÄ

Esineiden internettiin tarkoitetut sulautetut käyttöjärjestelmät ovat tarvittavia laitteiston ja sovellusten hallintaan IoT järjestelmissä. Saatavilla olevien IoT käyttöjärjestelmien joukosta osa on suurten yritysten tai instituutioiden tukemia, ja osa on täysin vapaan lähdekoodin yhteisön kehittämiä. IoT käyttöjärjestelmän testaus keskittyy esineiden internetin avainominaisuuksiin kuten verkkotietoliikenteeseen ja rajallisiin resursseihin.

Työssä tunnistetaan Mbed OS käyttöjärjestelmän testausmenetelmien ongelmia ja kehitetään yksikkötestaustyökalu. Kehitetty yksikkötestausympäristö mahdollistaa kehittäjille yksikkötestien kirjoittamisen ja ajamisen. Testaustyökalu yhdistetään myös Mbed OS jatkuvan integraation prosessiin testauskattavuuden parantamiseksi.

Työssä katsotaan kuinka funktionaaliset testit ja yksikkötestit ovat yleisimmät testityypit avoimen lähdekoodin sulautetuissa käyttöjärjestelmäprojekteissa. Mbed OS yksikkötestaustyökalu näyttää kuinka testien ajaminen PC ympäristössä on nopeampaa kuin IoT laitteissa. Tämä työkalu myös mahdollistaa kehittäjien kirjoittaa yksikkötestejä vapaammin ja siten parantaa kehitysprosessia.

Kehitetty yksikkötestaustyökalu ratkaisi Mbed OS testauksen ongelmia, mutta syventävää tutkimusta tarvitaan enemmän testausmenetelmien parantamiseksi edelleen.

Avainsanat: IoT, avoin lähdekoodi, yksikkötestaus, funktionaalinen testaus, regressiotestaus, jatkuva integraatio

TABLE OF CONTENTS

FOREWORD

This thesis is based on the work I did at Arm. I would like to thank my supervisors Dr tech. Susanna Pirttikangas and Dr tech. Teemu Leppänen for instructions during the writing period. Olli-Pekka Puolitaval acted as the technical supervisor for this thesis at Arm. He also helped with the scope and the focus of this thesis. I also want to thank my family. They have been very supporting of me during this process.

Oulu, 12th December, 2019

Lari-Matias Orjala

LIST OF ABBREVIATIONS AND SYMBOLS

API	Application Programming Interface
AWS	Amazon Web Services
BSP	Board support package
C	Programming language
C++	Programming language
CD	Continuous delivery
CI	Continuous integration
CLI	Command line interface
CPU	Central Processing Unit
FVP	Fixed virtual platform
Greentea	Mbed test automation tool
GUI	Graphical user interface
HTML	Hypertext Markup Language
IoT	Internet of Things
MCU	Microcontroller unit
PC	Personal computer
PR	Pull Request
OAT	Operational acceptance testing
OS	Operating system
RTOS	Real-time operating system
UAT	User acceptance testing
XML	Extensible Markup Language

1. INTRODUCTION

Internet of Things (IoT) means autonomous machines which are communicating with each other. It is estimated that by 2020 there would be around 30 billion connected devices [1]. The definition has been changing over the years since new applications have emerged which do not fit the original definition [2]. IoT applications such as smart homes and buildings, wearables, connected cars and smart cities projects among others are developed and deployed all around the world. The IoT market grows every day and more is invested in trying to find new leading applications for the consumer, commercial, industrial and infrastructure spaces.

Internet of Things applications have many requirements. The applications have to support various connectivity technologies, establish high security features and keep low power consumption. Devices can range from small microcontroller units (MCU) to high-performance systems so the software must be modular enough to be used by different systems with different amount of memory.

The operating system sits between the physical hardware layer and the application layer. It is used to manage hardware and software of the system. It provides an abstraction layer for the hardware it runs on so applications can utilize those resources. IoT operating systems are embedded operating systems for IoT applications. IoT operating system provides all the resources needed to run these applications. IoT operating systems focus on providing features such as security, the support of various connectivity technologies, small memory footprint and low power consumption.

There is a number of IoT operating system projects available and in active development. Some projects are backed by large companies or institutes and some are developed completely by the open source community. Closed source or commercial operating systems are found in systems where reliability and safety are expected and where professional support is needed. Open-source operating system projects have gained a lot of interest over the years because of free licensing models and active community support.

Open-source IoT operating system projects need to have a proper testing process in place in order to maintain quality. The testing of IoT operating systems share many challenges with the testing of any embedded system where the software is run in constrained hardware environment. This thesis explores the testing methods of multiple state-of-the-art open-source IoT operating system projects.

The goal of this thesis is to improve the testing methods of Mbed OS by implementing a unit testing framework for Mbed OS development and into Mbed OS continuous integration pipeline. This work was done for Arm which is why Mbed OS was selected. Mbed OS targets devices using Arm Cortex-M microprocessor core architectures which is why similar operating system projects targeting low-end devices with Cortex-M cores were selected for comparison.

2. BACKGROUND

IoT applications are software applications which are running on IoT devices. There can be many different kinds of IoT applications. Atzori et. al. present three different visions of IoT: **Internet oriented**, **Things oriented** and **Semantic oriented** perspectives [3]. Internet oriented vision focuses on networks of uniquely identifiable devices. Things oriented vision focuses on the device and different technologies which allow the devices to communicated with each other. Semantic oriented vision is the last perspective which tries to solve issues of representing, storing and organizing of information generated by the IoT. Applications can emphasize different visions which is why requirements can be different between different applications.

An embedded operating system is needed for IoT applications in order to run the software application on the IoT device. The applications have to work correctly so the quality of the operating system is important. Software testing is a process of evaluating the quality of the system. This process is not different for embedded operating systems for IoT. The testing of IoT operating systems follows the basic principles of testing any software system. The testing can happen in many forms during the software development process but proper testing process is a key to a quality product.

2.1. Internet of Things operating systems

An operating system is software which manages hardware components and allows the running of applications which utilize mentioned hardware and other system resources. The operating systems consist of components which are needed to run applications. The most important component is the kernel. **Kernel** is the core of an operating system and it is used to manage resources, memory and the hardware devices of the system. Kernel runs in a protected space of the system called kernel memory and it handles system calls from the software applications running in the user space and translates them into the instructions for the central processing unit (CPU).

Scheduler is used to decide which process can be running at the time. There are three types of schedulers: long-term, medium-term and short-term schedulers. The long-term scheduler selects which process is admitted into main memory and therefore for processing. The medium-term scheduler is used to temporarily swap processes out of main memory and into secondary memory or vice versa depending on the scheduling strategy. Medium-term scheduling helps to move inactive or blocking processes out of the memory to free up memory for other processes. Finally, the short-term scheduler is used to selecting which process is executed next and by which CPU.

Scheduling allows computer multitasking which means running multiple tasks over a certain period of time using a single CPU. Scheduling strategy for multitasking determines what type of operating system it is. **Cooperative multitasking** also known as non-preemptive multitasking is a multitasking strategy where the system does not perform context switching but the processes

themselves must allow other processes to run eventually. **Preemptive multitasking** is a strategy where the system can suspend the running process for other processes. This allows all processes to get some CPU time and handle process scheduling using the system scheduler. **Real-time operating system (RTOS)** is a operating system where the scheduling and execution time of the processes is guaranteed. This means the system is under time constraints and must execute processes preemptively using event priority or time-sharing. Event priority scheduling means higher priority events are executed first. Time-sharing scheduling means a period of execution time is allocated for each process by the scheduler.

Operating system often provides additional features such as device drivers which are used for allowing the operating system communicate with the hardware, file systems which allows data to be stored in files, and networking solutions which allow the applications to communicate with other applications and Internet.

IoT operating systems are embedded operating systems, which target IoT hardware. Embedded operating systems are operating systems which are designed to be efficient of available resources and reliable. IoT operating systems are often real-time operating systems since IoT applications require critical management of hardware and application resources under strict time constraints [4].

2.2. Testing of open source software

Testing is a process of validating the quality of the software and the code written for that software. Myers et. al. say that "software testing is a process, or a series of processes, designed to make sure computer code does what it was designed to do and, conversely, that it does not do anything unintended" which supports this idea that software testing is needed to make sure the code is working [5]. There are different approaches to testing IoT operating system code and different test environments to run tests.

2.2.1. Testing approach

Different software testing strategies give different results, and therefore it is important to choose proper testing methods for software validation and the development process. Testing approaches are often divided into passive and active testing.

Passive testing is a testing approach where the system is tested without actively interacting with it. Passive testing consists of system monitoring, the analysis of past test results and the analysis of logs.

Active testing is the testing approach where software is tested before each release. Test data is acquired from these tests and analysed. Active testing can be static or dynamic. Static testing means no test cases are executed. Static testing consists of static code analysis, reviews, walkthroughs and inspections where the goal is to find defects and errors as early as possible in the development process.

Dynamic testing means running test cases against the system. The size or complexity of the testable unit determines the level of dynamic testing. The common levels of testing are unit testing, integration testing and system testing. Testing is called unit testing when testing is done against the smallest unit of code such as single function. Each unit test tests only that single independent unit of code and communication between other units is often done by mocking or stubbing all external dependencies. Integration testing is when external dependencies are included in the tests and the interfaces between the system components are verified. Testing is defined as system testing when the complete system is verified against the product requirements. Figure 1 shows a testing pyramid by Mike Cohn [6]. The testing pyramid shows the relation between unit tests, integration tests and system tests. Tests higher in the pyramid require more integration and are slower to execute. In practice this means writing tests with different granularity and having fewer high-level tests [7]. High-level tests are tests which test the whole system instead of particular components. Acceptance testing is often included as the final level of testing. Acceptance testing can either be user acceptance testing (UAT) or operational acceptance testing (OAT). In user acceptance testing, the system is verified against user-defined test cases in real-world scenarios. Operational acceptance testing is the evaluation of operational readiness of the system.

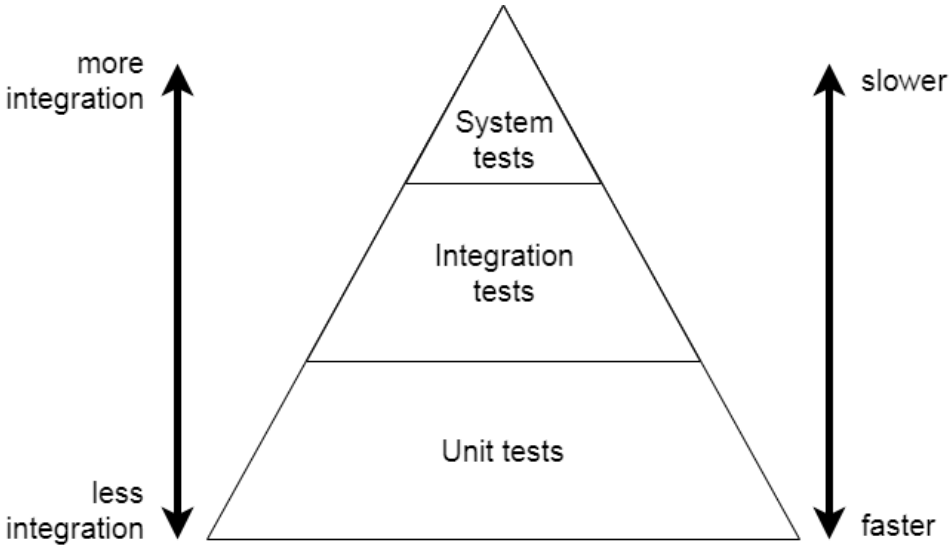

Figure 1. Testing pyramid

The approach to testing can also be exploratory or scripted. Exploratory testing combines designing and running tests with learning aspect so the tester can try different solutions in order to find as many problems in the system as possible. In scripted testing the tester follows the documented process of testing steps. This means the testing process is predefined and systematic.

Traditionally testing methods are separated into either white box or black box testing. White box testing means that the internal structures of the systems are used in the testing process. White box testing techniques are methods such as

API testing and code coverage. Black box testing is done from the perspective of an end user. In the black box testing, the internal structures of the system are unknown. The system accepts inputs and gives an output result. Gray box testing is a middle-ground between the two. In the gray box testing the tester has limited knowledge of the system.

The testing of IoT software focuses on specific features such as security, networking, device and platforms. The following types of testing are commonly used to test IoT software around specific features [8]:

- Usability testing
- Compatibility testing
- Reliability and scalability testing
- Data integrity testing
- Security testing
- Performance testing

Usability tests evaluate how the system works for different users. Different users require different features for the system so it is important to evaluate the system against particular use cases. Compatibility tests evaluate how different devices with the different hardware configurations can be connected using the IoT system. There are lots of IoT devices with different strengths and weaknesses so choosing the optimal device requires compatibility testing, Reliability and scalability tests focus on network and sensor performance. These tests are important for measuring how well the system scales for a large number of network devices. Data integrity testing evaluates how large amount of data, the application and the file system work with each other. IoT systems often transmit large amount of data and that data needs to be stored properly so ensuring that the data is accurate and consistent is important. Security tests work with data privacy controls and user or device authentication. Maintaining high security for data privacy and device or user identification reasons is essential. Finally, the performance testing which evaluates the performance of the IoT system. This means testing how well the system performs the tasks it was designed for.

Regression testing is a testing method where the tests are re-run to ensure that the software is working after changing the code. Regression tests are often functional and non-functional tests.

2.2.2. Test environments

IoT operating systems target specific hardware with limited resources. The modules and components of the operating system need to be tested on the actual IoT device to verify everything is working correctly. Testing only on the hardware is very costly and time-consuming. Tests need to be build for the specific hardware and executed in that environment. This is often near impossible especially in continuous integration (CI). There are however operating system modules which can be tested outside the hardware. Choosing correct tests to run on correct platform or test environment is essential in CI.

Testing on the actual IoT device is the most natural method of testing IoT operating system. Tests executed on the hardware verify that all aspects of the operating system work as expected. It is common to find issues or limitations only on specific platforms. Running tests on the device first requires building a test application image. Test application is an application which includes the operating system modules and tests which test those modules. The application is built into a binary image and which is then transferred into the device flash memory in operation called "flashing". When the application is loaded into memory after the reboot, the tests are then executed on the device.

A simulator environment tries to simulate the targeted hardware platform. Testing on simulated hardware does not require to own or purchase real devices. Simulators can be used to test many features, but they cannot simulate precise timers, real-life conditions and actual real hardware reliably [9]. The simulator is running on PC hardware and on operating systems such as Linux or Windows which are general purpose operating systems. These operating systems cannot simulate the strict timing requirements of RTOS so real IoT hardware is needed for tests which require these kind of requirements.

Native PC platforms running x86 architecture such as Windows, Linux and Mac OS are great for testing features with no concern of limited resources. Native or virtual computers for these operating systems provide speed and reliability to testing. Native computer environments are great for unit tests since unit tests do not often require specific hardware and developers can run unit tests in the middle of development.

Testing on PC platforms requires compiling the tests to x86 architecture. GNU tools such as GNU Compiler Collection (GCC) and GNU Make can be used to compile C/C++ to x86 architecture. GCC provides compilers for C and C++ languages and Make is used for generating the test executables by using the compilers. However, the operating system must support compiling to these platforms. GNU tools are available natively on Linux and Mac OS, and on Windows through MinGW. CMake tool can be used to provide cross-platform build environment for all the mentioned platforms. CMake provides compiler-independent build configurations which can be built using the compilers available on the running platform.

2.2.3. Test automation and continuous integration

Manual testing is slow and often impossible to do for every revision of developed software. Automated software testing can easily test every system iteration. Test automation means running tests on a separate software from the testable system. Huizinga et. al. present this as separation between test execution and software to be tested and the comparison of real test results with the expected outcome [10].

There are multiple approaches to test automation such as the graphical user interface (GUI) testing, the application programming interface (API) driven testing, framework testing and continuous testing. Automation frameworks include data-driven testing, modularity-driven testing, keyword-driven testing,

hybrid testing, model-based testing, code-driven testing and behaviour driven testing.

Test automation for embedded operating system projects means automating unit tests and regression tests in continuous integration and delivery (CI/CD). Regression or smoke testing means running the existing set of functional tests against the new software version in order to validate no issues arise. Continuous integration is a pipeline which automates the testing process by running tests for new software versions. A pipeline is a process which contains a start and an end and all the steps between needed to reach the end. The CI pipeline can be set to trigger on schedule or by an event such as Pull Request (PR). Open-source projects usually have PR validation enabled which means running predefined checks against the PR. Daily or nightly CI pipelines are also common to execute tasks which might take too long for PR pipeline.

Figure 2 shows the process of making a PR. Pull request is created in the first step. This allows the review process and the CI process to begin. Appropriate people are assigned for reviewing the PR. They go through the code and decide if changes are needed or not. While the review process is in place the code is tested for each PR check. PR checks are defined beforehand and they can be static code analysis, code style analysis, compile checks, tests or many other tasks which can be automated and can be used to validates the PR. If a required PR check fails or changes are requested then the PR goes through the check and review process again until the PR is in such state that it can be merged to the code base by the maintainers.

There are many continuous integration tools available. Table 1 shows top 9 used CI software with free license according to online surveys [11]. Jenkins is open source CI software which allows free on premise CI setup. The majority of CI software are hosted online services, but often they provide enterprise solutions for on premise installations.

Table 1. Top 9 free CI software in 2019

Software	Maintainer	License
Jenkins	community	MIT
Travis CI	Travis CI	Free for open-source, Proprietary
TeamCity	JetBrains	Limited free, Proprietary
CircleCI	CircleCI	Limited free, Proprietary
Codeship	Codeship	Limited free, Proprietary
Gitlab CI	Gitlab	Limited free, Proprietary
Buddy	Buddy	Limited free, Proprietary
Wercker	Oracle Corporation	Limited free, Proprietary
Shippable	Shippable	Free for open-source, Proprietary

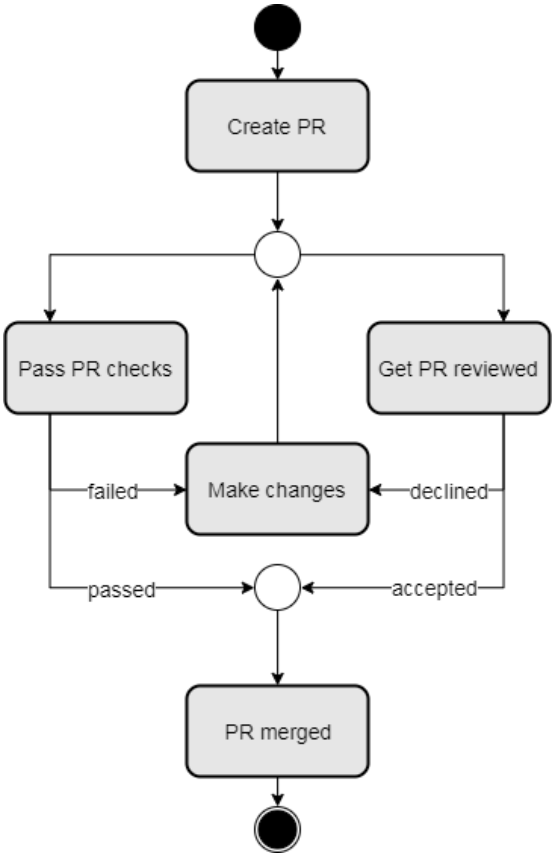

Figure 2. Pull request process

2.2.4. Unit testing frameworks

A unit testing framework is a collection of tools for helping to write and run unit tests. Unit testing frameworks are often categorized into xUnit derivative frameworks and other frameworks. A framework is considered xUnit derivative if it implements the test structure and execution design originally popularized by SUnit unit testing framework for Smalltalk programming language. Unit testing frameworks which are xUnit derivative use test architecture which consists of test suites, test cases, test fixtures, assertions and a test runner. Other frameworks use other test architectures and methods in order to run unit tests.

Assertions are used to test a single logical condition. They are the smallest macro of xUnit frameworks and verify the unit under test. **Test case** is a single test in xUnit frameworks and can contain multiple assertions. Test cases have a name which can be used to identify which test case failed or passed. **Test suites** are a collection of test cases. Test suites are used for similar test cases or when the test cases need to share test code. Order of tests does not matter in a test suite since all test cases are verified. When test cases share code or preconditions, **test fixtures** can be used. The test fixture sets up the initial state for the common code which then can be used by many test cases. **Test runner** is a program which can be used to run all written tests. XUnit frameworks usually also provide a test report formatter which outputs test results in many formats such as XML.

Test case execution in xUnit-based tests consists of three steps. The first step is **setup**. In setup, an isolated test environment is prepared for the test. After setup, the test is then run. The final step is **teardown** where the test environment is destroyed regardless of the test resolution.

There are many open source unit testing frameworks available. The most feature-rich state-of-the-art unit testing frameworks for C/C++ projects are listed in the Table 2. All the frameworks except Catch2 are xUnit derivatives and follow the principles of xUnit frameworks. Catch2 uses require and section macros instead of test fixtures. Catch2 also supports behaviour-driven development (BDD) macros to write tests using BDD terminology.

Table 2. List of state-of-the-art open-source C/C++ unit testing frameworks

Framework	License
Boost.Test	BSL-1.0
Catch2	BSL-1.0
CppUnit	LGPL
CppUTest	BSD-3
Google Test	BSD-3
Unity	MIT

Most unit testing frameworks also support parameterized tests or data generators. Parameterized tests allow the same test to be run multiple times using different values. Data generators generate data which is used for test values.

Mocks and stubs can be used to test code without external dependencies. Mocks and stubs act as the external dependencies but without actual implementation. The stub is a minimal presentation of code which actual compiles. Stubs can be classes or functions which implement the required interface without functionality. Mocks are way to preemptively assign an expectation to a call. This means a mocked call is assigned expected behavior or return value that expectation is determined when the code is run. All C/C++ code support stubbing, but test frameworks must provide the support for mocks. Google Test, CppUTest, Boost.Test and Unity provide support for mocks through an additional module.

2.2.5. Evaluation of testing methods

Evaluation of IoT operating system testing methods is different than measuring actual tests themselves. Unit testing metrics are a subset of test metrics. These metrics can be used to measure the quality of the unit tests and unit testing tools. Table 3 lists useful metrics for evaluating unit tests.

Testing metrics can be used to measure tests, but other metrics are needed to measure tools used for software testing. Michael et. al. present a set of objective metrics which can be used for measuring the effectiveness of software testing tools

Table 3. Unit testing metrics

Metric	Description
Unit test Volume	Number of unit tests.
Code coverage	Percentage of code tested.
Runtime	Time to run all unit tests.

[12]. These metrics can be used to measure the effectiveness of the testing tools of IoT operating systems which are implemented.

Human Interface Design (HID) measures the difficulty of using the tool and how likely there will be errors when using the tool for a long time. High HID value usually means there are issues in the user interface. HID is calculated as

$$HID = KMS + IFPF + ALIF + (100 - BR), \tag{1}$$

where the variables are

KMS = the average number of keyboard to mouseswitches per function,

$IFPF$ = the average number of input fields per function,

$ALIF$ = the average string length of input fields, and

BR = the percentage of buttons whose functions were identified via inspection by first time users times ten.

Maturity & Customer Base (MCB) measures how mature the tool is. MCB is calculated as

$$MCB = M + CB + P, \tag{2}$$

where the variables are

M = the number of years the tool has been used in real world software,

CB = the number of customers with more than one year of experience applying the tool, and

P = the number of previous or other projects that used the tool.

Tool Management (TM) measures the level of access to information or functions. TM is calculated as

$$TM = AL + ICM, \tag{3}$$

where the variables are

AL = the number of access levels to tool information, and

ICM = the sum of the different methods of controlling tool and test information.

Ease of Use (EU) measures how easy it is for new users to learn how to use the tool, how easy it is to retain knowledge for frequent and casual users and what is the operational time for frequent and casual users. EU is calculated as

$$EU = LTFU + RFU + RCU + OTFU + OFCU, \tag{4}$$

where the variables are

$LTFU$ = the learning time for first time users,

RFU = the retainability of procedure knowledge for frequent users,

RCU = the retainability of procedure knowledge for casual users,

$OTFU$ = the average operational time for frequent users, and

$OFCU$ = the average operational time for casual users.

User Control (UC) means how well the tool allows users to specify which areas of the software are critical to test or need more test coverage. High UC values mean the tool gives freedom to testers.

Test Case Generation (TCG) measures how easy it is to automatically create or generate or modify test cases. TCG is calculated as

$$TCG = ATG + TRF, \tag{5}$$

where the variables are

ATG = the level of automated test case generation between 0 and 10 where 0 is test generation by hand and 10 is fully automated test case generation, and

TRF = the level of test case reusability between 0 and 10 where 0 means test cases cannot be modified and 10 means test cases can be modified by user friendly methods.

Tool Support (TS) measures how much technical support and documentation is available to the users. TS is calculated as

$$TS = ART + ARTAH + ATSD - DI, \tag{6}$$

where the variables are

ART = the average response time during scheduled testing,

$ARTAH$ = the average response time outside of scheduled testing,

$ATSD$ = the average time to search documentation for desired

information, and

DI = the number of unsuccessful searches of documentation.

Estimated Return on Investment (EROI) measures how much benefit is expected to get by using the tool. EROI is calculated as

$$EROI = (EPGxETTxACTH) + EII - ETIC + (EQCxEHCSxACCS), \quad (7)$$

where the variables are

EPG = the estimated productivity gain,
ETT = the estimated testing time without tool,
$ACTH$ = the average cost of one testing hour,
EII = the estimated income increase,
$ETIC$ = the estimated tool implementation cost,
EQC = the estimated quality gain,
$EHCS$ = the estimated hours of user support per project, and
$ACCS$ = the average cost of one hour of user support.

The rest of the metrics are **Reliability (Rel)** which is the mean time between failures, **Maximum Number of Classes (MNC)** which is the maximum supported number of classes in a project, **Maximum Number of Parameters (MNP)** which is the maximum supported number of parameters in a project, **Response Time (RT)** which is the amount of time to implement a test case for a project, and **Features Support (FS)** which accounts features such as extensibility, integration with development tools and reporting.

Acceptance and portability tests which are used to evaluate how well a new tool will fit into the testing process. The acceptance tests measure how well the tool meets the requirements. Portability tests are used when there are old tests or tools and it needs to be measured how easily the old tests or tools can be converted to use the new tool.

2.3. Open-source operating system projects

There are many open-source IoT operating system projects available to use or contribute to. The projects usually have a regression test process in place and often even provide custom or third-party testing tools for helping with testing. Qutqut et. al. provided a great list of operating systems for low-end IoT devices [13]. I updated this list to include current state-of-the-art operating systems which support Arm Cortex-M based IoT devices. Table 4 lists current state-of-the-art open source operating systems. These operating systems and their testing approaches are introduced next.

Apache Mynewt is an open source real-time operating system managed by Apache Software Foundation. Apache Mynew targets devices using MCUs from Arm Cortex-M0-M4 family, MIPS and RISC-V. Apache Mynewt is a full

Table 4. State-of-the-art open-source operating systems for low-end IoT devices

Operating system	Managed by	License
Apache Mynewt	Apache Software Foundation	Apache License 2.0
Contiki-NG	Contiki-NG maintainers	BSD-3
Amazon FreeRTOS	Amazon Web Services	MIT
Mbed OS	Collaborative project managed by Arm	Apache License 2.0
RIOT	Freie Universität Berlin, INRIA, and Hamburg University of Applied Sciences	LGPLv2
Zephyr	Linux Foundation, Wind River Systems	Apache License 2.0

operating system and features a kernel, drivers, file system and a networking stack of various technologies and protocols [14].

Apache Mynewt provides a tool for build and package management called Newt. Newt tool can also be used to run unit tests. Apache Mynewt also provides a tool for testing packages on embedded hardware or in simulated environment called testutil. Unit tests are found inside the main repository next to the features they are testing in separate folders. Apache Mynewt uses Travis CI to validate Github Pull Requests (PR). Pull request validation consists of building core application for 31 device platforms, building blinky application (application which demonstrates LEDs on the device) using 60 different Board Support Packages (BSP) and finally running unit tests with Mynewt tool on Linux and Mac OS.

Contiki-NG is a fork of a popular open source multitasking operating system Contiki [15]. Contiki-NG targets devices from Arm Cortex-M3-M4, TI MSP430, TI SimpleLink and other 32-bit MCUs. Contiki-NG supports native Linux, Mac OS and Windows processes and simulation with Cooja virtual platform. Contiki-NG is a full operating system and features a kernel, drivers, file system and a networking stack of various technologies and protocols.

Contiki-NG applications are built using a selection of 3rd party development tools, but provides a Docker image for easy development environment installation using Docker virtualization. Tests are located in the main repository under the tests directory. Unit tests are written using their own unit testing tool. The tool is a C/C++ script which allows defining, registering and running unit tests. Contiki-NG uses Travis CI to validate Github Pull Requests. CI runs tests by executing make in each test directory. CI builds documentation, compiles example applications for native PC platform and various IoT boards, runs simulation tests and unit tests.

FreeRTOS is an open source real-time operating system kernel originally developed by Richard Barry and his company Real Time Engineers Ltd [16]. The FreeRTOS project was passed to Amazon Web Services team in 2017 [17]. Amazon provides a commercial version of Amazon FreeRTOS called OpenRTOS

sold by Wittenstein High Integrity Systems (WHIS). WHIS also develops safety-certified SAFERTOS which is based on FreeRTOS, but targets safety-critical markets such as medical, industrial and automotive fields. FreeRTOS supports over 40 different MCU architectures such as Arm Cortex-M0-M33 and RISC-V including over 15 different compiler toolchains to build applications for these architectures. FreeRTOS is a kernel-only operating system and provides no drivers, file system or networking features by default. Amazon FreeRTOS extends FreeRTOS kernel with connectivity, security and over-the-air (OTA) updates.

FreeRTOS kernel only provides demo applications publicly for testing FreeRTOS ports. Tests for these port features are located in the code repository and have to be run on the actual hardware. Older common code is tested using SAFERTOS test harness and new features have tests which have not been published [18]. Amazon FreeRTOS however provides tests in their repository for modules outside the kernel. Amazon FreeRTOS also provides AWS IoT Device Tester tool to build, flash and run tests. Amazon FreeRTOS uses Unity unit testing library for internal unit tests. Amazon FreeRTOS CI is not public, but it contains build and code style checks, building on Linux and 97 AWS Batch jobs which test different features of the operating system.

Mbed OS is an open source embedded operating system designed for IoT systems [19]. Mbed OS is part of Mbed platform managed by Arm. Mbed OS is a full operating system with RTOS, file system, security and a networking stack of various technologies and protocols. Mbed OS 5.12 is also PSA certified [20]. Mbed OS targets devices running Arm Cortex-M family MCUs. Mbed OS is part of Mbed secure device-to-cloud story with seamless integration with Pelion Device Management Update Services which provides remote firmware updates.

Mbed OS provides Mbed CLI tool for building applications, flashing and testing. Mbed CLI can be integrated with all supported toolchains: Arm GCC, Arm Compiler and IAR. Mbed also provides Online compiler for building applications. For testing Mbed OS provides Python-based testing tools [21]. Greentea is a tool for running functional unit or integration tests on hardware. Greentea uses utest test harness to run tests on the device. Greentea works by automatically building, flashing and running the tests on a single IoT device. Icetea allows interoperability testing using multiple devices and external services. Icetea works by flashing a command-line based application to the device and then sending test commands from the PC to the device through serial terminal. Mbed unit testing tool is the tool developed in this thesis work and used for testing Mbed OS components on PC environment. Mbed OS can also be simulated for Cortex-M-based Fixed Virtual Platforms (FVP) using Arm FastModels. Greentea tests can be run on FastModel FVPs using the mbed-fastmodel-agent Python module. Mbed OS tests are located in the code repository. Greentea tests are located under TESTS, Icetea tests under TEST_APPS and unit tests under the UNITTESTS directory. Mbed OS CI uses both Jenkins and Travis CI for validating Github Pull Requests. Travis CI is used for building documentation, testing Python tools, testing the littlefs file system and other small tasks such as license check. Jenkins is used for running unit tests, building test applications, running hardware tests, checking dynamic memory usage and other more tasks which are heavy on

resources. The Jenkins pipeline adjusts to the content of the Pull Request so only necessary tasks are run [22].

RIOT is an open-source IoT operating system originally developed by FU Berlin, INRIA and HAW Hamburg [23]. RIOT is based on FeuerWare, an operating system for Wireless Sensor Networks [24]. RIOT was originally named μkleos but was renamed in 2013 when the project went public. RIOT targets devices from AVR, ARM7, Cortex-M0-M7, Cortex-M23, ESP8266, ESP32, MIPS32, MSP430, PIC32, RISC-V and x86 architectures. RIOT is a full operating system with a kernel, drivers, file system and a networking stack which supports various technologies and protocols.

RIOT tests are located in the repository under tests. RIOT provides test applications to be run on hardware. Tests are run by running make inside the test directory. The test runner for RIOT is Python-based tool. Unit tests are also provided and Embedded Unit unit testing framework is used to run unit tests on hardware [25]. RIOT CI uses Travis CI and Murdock for validating Github Pull Requests. Travis CI is used for static code analysis and building documentation. Murdock is a CI server written in Python and developed for RIOT [26]. Murdock is used for running static tests, building test applications and running hardware tests.

Zephyr is an open source RTOS for devices with constrained resources hosted by Linux Foundation [27]. Zephyr supports MCU architectures such as ARC, Arm, NIOS II, RISC-V, Tensillica and x86. Zephyr is a full operating system with a kernel, drivers, file system and networking stack which supports various technologies and protocols.

Zephyr provides a command-line tool for source code management, flashing and debugging called west. For testing Zephyr provides a test framework called Ztest (Zephyr Test Framework). Ztest can be used for integration tests and unit tests. Tests are located in the repository under tests. Unit tests can be run on the QEMU emulator or on real hardware by using the sanitycheck script. Sanitycheck allows running tests on a single device or multiple devices. Zephyr uses Shippable for CI. Zephyr CI validates Pull Requests by checking Git commit formatting and coding style (linting), building documentation and running sanity check tests for various boards and MCU architectures in emulated environment. Zephyr also runs nightly builds against the master branch on Shippable.

3. DESIGN OF UNIT TESTING

The testing of Internet of Things software share many methods and ideologies that are used in other software testing especially in embedded software testing. Similarly common issues can be identified from these testing processes which guide the testing methodology towards better results. IoT operating system testing methods follow these widely used practises, but set the focus on the number of supported target platforms and validation of management of limited platform resources. This chapter lists problems in Mbed OS testing methods, finds solutions to fix those problems and presents a unit testing framework implementation for Mbed OS.

3.1. Identifying problems in Mbed OS testing methods

Identifying problems in testing methods is a common task in any software development process. Common testing problems can be identified since many testing methods are similar in IoT operating system projects. Testing problems can be generic or affect only certain type of testing or testing of the particular component [28].

Common testing methods for the operating systems from the previous chapter show that there are mainly two kinds of tests in open source IoT operating system projects:

- Functional tests
- Unit tests

Table 5. Capabilities of testing methods for each OS

OS	Unit tests			Functional tests		
	x86	simulator	HW	x86	simulator	HW
Apache Mynewt	no	yes	yes	no	yes	yes
Contiki-NG	yes	yes	yes	yes	yes	yes
Amazon FreeRTOS	no	no	yes	no	no	yes
Mbed OS	yes[1]	yes	yes	no	yes	yes
RIOT	no	no	yes	no	no	yes
Zephyr	yes	yes	yes	yes	yes	yes

[1)] Mbed OS unit testing framework was developed for this thesis

These types of tests are usually written for operating systems. Functional tests (test applications) and unit tests are often run as regression tests in addition to building tests. Additional dimensions are the testing platform and continuous integration. Functional tests are mainly run on real IoT hardware. Unit tests are run on x86 platforms, simulators and even in hardware. Different projects run

tests on different environments. In CI testing mainly consists of running tests on x86 or simulator environments. Mbed OS and RIOT run hardware tests in CI. Table 5 summarizes the capabilities of testing methods on studied OSes. We can see from the table which testing methods are missing and try to implement them.

The main problems in Mbed OS testing were caused by long Pull Request flow in Mbed OS CI pipeline. The reason for this was no unit tests, no auto triggering tests which means no tests which are automatically started when PR is created or updated, limited test resources and manual Pull Request process. The problem with Mbed OS regression test pipeline was that all tests were Greentea hardware tests which means tests are Mbed OS applications which have to be built and then executed in limited number of IoT devices. Running tests on real hardware is slow and costly.

3.2. Solutions to Mbed OS testing

Mbed OS CI related problems were mostly solved by improving Mbed OS regression test selection and optimization, but it did not solve all the problems in Mbed OS testing [22]. Mbed OS needed lower level tests. There was no proper unit testing process for Mbed OS testing in place until Mbed OS 5.10. Mbed OS had unit tests for hardware and some networking unit tests for PC platform, but no single framework to rely on and no common process to follow. No unit tests were executed for PRs in the regression pipeline either. Fast Models could also be added to run tests on the simulated environment. Using a simulator instead of actual hardware would extend hardware life and reduce test resource costs.

3.3. Requirements

Requirements for this unit testing framework were the following:

- C/C++ compatible
- Cross-platform support
- Mbed CLI integration
- Easy CI integration
- Standard test report formats
- Unit tests are easy to add and extend
- Parameterized tests
- Mocks can be used

Mbed OS codebase is 94.1% C and 3.6% C++ which means that C/C++ compatible tools are required. The unit testing framework has to be compatible with all major development platforms: Windows, Mac OS and Linux. How the tool is used has to be nearly identical regardless of the development platform. Unified user interface and Mbed CLI development tool integration are required.

The framework must be easy to install and integrate into a testing pipeline. Unit test results have to be repeatable so this means the tool and all its

dependencies must be under version control. In CI test reports are collected and analysed which means the generated test reports have to be in computer-readable text format. XML and HTML are the most commonly used markup languages today which makes them ideal for test reporting.

The process of adding new unit tests has to be simple. First, unit tests should be written with minimal amount of configuration. Too much scaffolding around tests increases unnecessary complexity. Tests should also auto-register so there is no need to keep a list of unit tests in code. The unit test framework should also support for parameterized tests so tests can be executed multiple times with different parameters.

3.4. Unit testing framework

A tool review was constructed in order to find the best unit testing framework for Mbed OS. Summary of the tool review can be seen in Appendix 1. I compared the features of the most feature-rich state-of-the-art open source C/C++ unit testing frameworks available listed in the previous chapter. Selection criteria for the tool were the number of features and active development of the framework. There were unit tests inside the Mbed OS repository which were using CppUTest. Google Test was selected as the underlying assertion library for the unit testing framework. Unit tests written use setup-teardown architecture shown in the Figure 3.

```
#include "gtest/gtest.h"
#include "platform/CircularBuffer.h"

class TestCircularBuffer : public testing::Test {
protected:
    mbed::CircularBuffer<int, 10> *buf;

    virtual void SetUp()
    {
        buf = new mbed::CircularBuffer<int, 10>;
    }

    virtual void TearDown()
    {
        delete buf;
    }
};

TEST_F(TestCircularBuffer, constructor)
{
    EXPECT_TRUE(buf);
}
```

Figure 3. Unit test for CircularBuffer

3.5. Building tests with CLI

Google Test acts as a test runner, but in order to build these test suites a build environment is needed. Compilers which can be used to build Mbed OS applications do not work on native development platforms. GNU compilers can be used instead since they can be installed on Windows, Linux and Mac OS. In addition, Arm GCC compiler is based on GNU GCC compiler which makes the transition easier. Make and g++ can be used to build the test applications.

CMake was selected as a cross-platform tool for setting the build environment on any platform. I configured CMake to automatically download Google Test and add it to each build target. Build targets means the test executables which are built using make. Using input variables for CMake it was possible to compile test applications optionally with code coverage flags. CMake toolkit includes CTest test runner which can be used to automatically find all built tests and run them. All unit test suites must have a configuration file for CMake. This configuration file is used to include necessary headers, test sources and stubbed sources to the built test. Figure 4 shows how the file is structured.

mbed-os ▸ UNITTESTS ▸ features ▸ cellular ▸ framework ▸ AT ▸ at_cellularbase ▸ unittest.cmake

```
1
2   ####################
3   # UNIT TESTS
4   ####################
5
6   # Add test specific include paths
7   set(unittest-includes ${unittest-includes}
8     features/cellular/framework/AT/AT_CellularBase
9     ../features/cellular/framework/AT
10    ../features/cellular/framework/common
11    ../features/frameworks/mbed-trace
12    ../features/frameworks/nanostack-libservice/mbed-client-libservice
13    ../features/netsocket
14  )
15
16  # Source files
17  set(unittest-sources
18    ../features/cellular/framework/AT/AT_CellularBase.cpp
19  )
20
21  # Test files
22  set(unittest-test-sources
23    features/cellular/framework/AT/at_cellularbase/at_cellularbasetest.cpp
24    stubs/mbed_assert_stub.cpp
25    stubs/ATHandler_stub.cpp
26    stubs/EventQueue_stub.cpp
27    stubs/FileHandle_stub.cpp
28    stubs/ConditionVariable_stub.cpp
29    stubs/Mutex_stub.cpp
30  )
```

Figure 4. Unit test configuration file for CMake

A Python-based command line tool was developed to put the building and running of tests behind a user-friendly interface. Python-based tool allows easy integration with Mbed CLI application. Figure 5 shows the architecture of the framework.

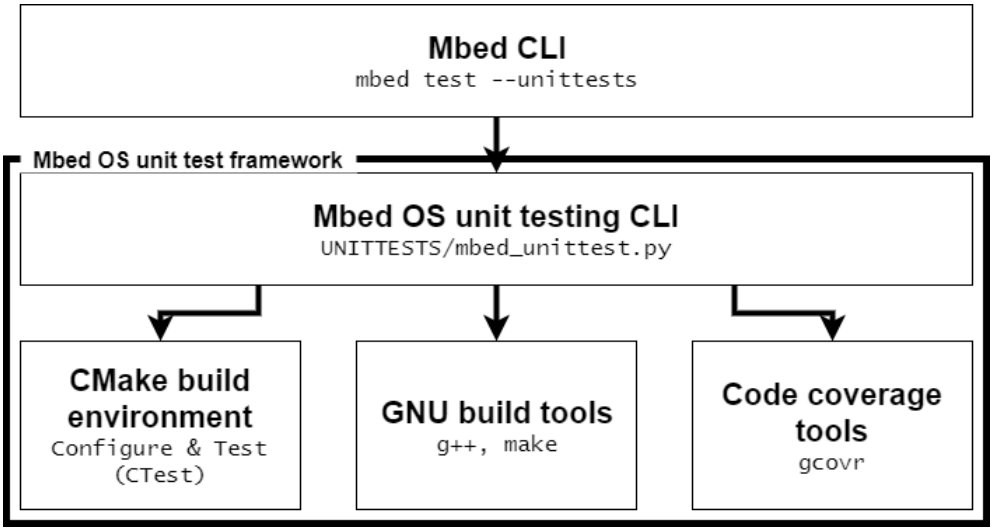

Figure 5. Mbed OS unit testing framework architecture

```
$ ./UNITTESTS/mbed_unittest.py --help
usage: mbed_unittest.py [-h] [-v | --quiet] [--compile] [--run] [-c] [-d]
                        [--coverage {html,xml,both}] [--include-headers]
                        [-m {gmake,make,mingw32-make,ninja}]
                        [-g {Unix Makefiles,MinGW Makefiles,Ninja}]
                        [-r TEST_REGEX] [--build BUILD] [--new FILEPATH]

optional arguments:
  -h, --help            show this help message and exit
  -v, --verbose         Verbose output
  --quiet               Quiet output
  --compile             Only compile unit tests
  --run                 Only run unit tests
  -c, --clean           Clean the build directory
  -d, --debug           Enable debug build
  --coverage {html,xml,both}
                        Generate code coverage report
  --include-headers     Include headers to coverage reports, defaults to
                        false.
  -m {gmake,make,mingw32-make,ninja}, --make-program {gmake,make,mingw32-make,ninja}
                        Which make program to use
  -g {Unix Makefiles,MinGW Makefiles,Ninja}, --generator {Unix Makefiles,MinGW Makefiles,Ninja}
                        Which CMake generator to use
  -r TEST_REGEX, --regex TEST_REGEX
                        Run tests matching a regular expression
  --build BUILD         Build directory. Default: UNITTESTS/build/
  --new FILEPATH        Source file from which to generate test files. E.g.
                        rtos/Semaphore.cpp
```

Figure 6. Mbed OS unit testing CLI

The unit testing tool provides options for configuring how CMake builds the tests. The tool can also be used to generate a new unit test suite from a template. Figure 6 shows how the command-line tool is used and all the options the tool allows to select.

3.6. Mbed OS CI integration

The unit testing framework was integrated into Mbed OS CI for running unit tests for Pull Requests. CI environment used is Jenkins. Unit tests run in Jenkins pipeline job written using Groovy. The job is integrated into the main pipeline which starts the job if unit tests need to be run.

There are three main steps in the job. The first step is building unit tests. The job sets a Python virtual environment and compiles all the unit tests with the code overage flag on. The second step is for running the tests using the unit testing tool while generating code coverage reports in HTML and XML. Tests are also analysed with Valgrind for memory leaks. The final step is publishing test results in xUnit format and code coverage reports. Figure 7 shows this test job in pseudocode.

```
def unitTest() {
  try {
    // Clean workspace

    // Clone Mbed OS

    dir("mbed-os") {
      stage("Build") {
        // Setup Environment

        sh "python ./UNITTESTS/mbed_unittest.py --compile --coverage both"
      }

      stage("Test") {
        sh "python ./UNITTESTS/mbed_unittest.py --run --coverage both"

        // Analyze with Valgrind
      }

      stage("Publish") {
        // Publish xUnit reports

        // Publish code coverage reports
      }
    }
  } catch (error) {
    // Throw error
  } finally {
    // Clean workspace
  }
}
```

Figure 7. Unit testing pipeline job

4. EVALUATION

The unit testing framework was developed for Mbed OS and unit tests were added into the Mbed OS CI for PR regression testing. The implemented framework was evaluated with different tests and an effectiveness analysis. Acceptance and portability testing was done in form of running existing unit tests with the framework and analysing results. Performance was tested against Greentea hardware tests. Here are the evaluation and the findings of implementing this method into Mbed OS testing. Data is acquired from Mbed OS code repository and Mbed OS continuous integration.

4.1. Acceptance and portability tests

The framework was tested by taking a subset of existing unit tests written for CppUTest. These tests were ported to Google Test format. Both frameworks use xUnit test structure so porting was a simple process. First included headers were updated by including Google Test instead of CppUTest test harness as seen in the Figure 8.

```
-#include "CppUTest/TestHarness.h"
 #include "test_at_cellularbase.h"
 #include "AT_CellularBase.h"
+#include "gtest/gtest.h"
 #include "test_at_cellularbase.h"
 #include "AT_CellularBase.h"
```

Figure 8. Include Google Test

Google Test uses test fixtures to define test suites for test cases which use the same data configuration. Ported tests all use setup-teardown structure so CppUTest TEST_GROUP for the testable classes were changed to test fixture classes. A test fixture class is shown in the Figure 9.

```
-TEST_GROUP(AT_CellularBase)
-{
-    Test_AT_CellularBase* unit;
-    void setup()
+class TestAT_CellularBase : public testing::Test {
+protected:
+    Test_AT_CellularBase *unit;
+    virtual void SetUp()
     {
         unit = new Test_AT_CellularBase();
     }
-    void teardown()
+    virtual void TearDown()
     {
         delete unit;
     }
 };
```

Figure 9. Create a fixture class

The process of updating the test case and assertion macros was the final step. Google Test uses TEST_F macro for the test fixture test case seen in the Figure 10. Assertion macro for equality comparison in Google Test is EXPECT_EQ which expects two values. Assertion macro changes from CppUTest to Google Test can be seen in the Figure 11.

The existing unit tests were then run using the unit testing framework to verify that test results were identical when using either library. These tests gave confidence that the framework could be added into Mbed OS testing tools.

```
TEST(AT_CellularBase, Create)
{
    CHECK(unit != NULL);
}

TEST(AT_CellularBase, test_AT_CellularBase_get_at_handler)
{
    unit->test_AT_CellularBase_get_at_handler();
}

TEST(AT_CellularBase, test_AT_CellularBase_get_device_error)
{
    unit->test_AT_CellularBase_get_device_error();
}
```

```
TEST_F(TestAT_CellularBase, Create)
{
    EXPECT_TRUE(unit);
}

TEST_F(TestAT_CellularBase, test_AT_CellularBase_get_at_handler)
{
    unit->test_AT_CellularBase_get_at_handler();
}

TEST_F(TestAT_CellularBase, test_AT_CellularBase_get_device_error)
{
    unit->test_AT_CellularBase_get_device_error();
}
```

Figure 10. Update test case macros

```
void Test_AT_CellularBase::test_AT_CellularBase_get_at_handler()
{
    EventQueue eq;
    FileHandle_stub fh;
    ATHandler ah(&fh, eq, 100, ",");
    AT_CellularBase at(ah);

    CHECK(&ah == &at.get_at_handler());
}

void Test_AT_CellularBase::test_AT_CellularBase_get_device_error()
{
    EventQueue eq;
    FileHandle_stub fh;
    ATHandler ah(&fh, eq, 0, ",");
    AT_CellularBase at(ah);

    ATHandler_stub::device_err_value.errCode = 8;

    CHECK_EQUAL(8, at.get_device_error().errCode);

    ATHandler_stub::device_err_value.errCode = 0;
}
```

```
void Test_AT_CellularBase::test_AT_CellularBase_get_at_handler()
{
    EventQueue eq;
    FileHandle_stub fh;
    ATHandler ah(&fh, eq, 100, ",");
    AT_CellularBase at(ah);

    EXPECT_EQ(&ah, &at.get_at_handler());
}

void Test_AT_CellularBase::test_AT_CellularBase_get_device_error()
{
    EventQueue eq;
    FileHandle_stub fh;
    ATHandler ah(&fh, eq, 0, ",");
    AT_CellularBase at(ah);

    ATHandler_stub::device_err_value.errCode = 8;

    EXPECT_EQ(8, at.get_device_error().errCode);

    ATHandler_stub::device_err_value.errCode = 0;
}
```

Figure 11. Update test assertion macros

4.2. Performance tests

Critical information was to measure the performance of the framework against existing tools. Greentea is used to run unit tests on Mbed hardware so these performance tests were done to know how well the unit testing framework performed against hardware tests.

The test was run on Xubuntu 18.04 LTS virtual machine on Apple MacBook Pro 13" 2017. Mbed device for this test was NXP FRDM-K66F which is a low-cost development platform running Arm Cortex-M4 processor cores. FRDM-K66F was connected to the laptop with a micro-USB-cable. Test setup is pictured in the Figure 12.

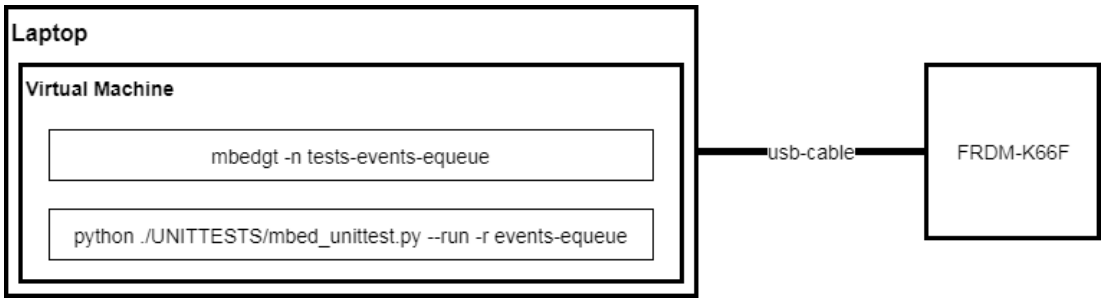

Figure 12. Performance test setup

One test case was used. The test case was the same for both unit testing framework and Greentea. Mbed OS event queue test which tests that event queue executes a function passed by equeue_call. Greentea test application included only the test_equeue_simple_call test case and Mbed OS unit tests only included the TEST_F(TestEqueue, test_equeue_simple_call) test case.

Performance was measured by running both unit testing tool and Greentea inside a timed function sequentially. One run was made using the unit testing framework CLI directly and one run was made using Mbed CLI. The tools also included the test case run-time. Mbed OS unit testing framework performance against Greentea when running the same unit test can be seen in the Table 6. These results show that unit tests run much quicker on the PC development platform. Test applications have to be flashed into the device to run hardware tests which is a slow operation. The same test case takes six times slower on this IoT hardware than running on PC. The tools themselves add tens of seconds to the overall runtime depending on how much abstraction there is and how many tools are used.

Mbed OS unit tests were enabled on Mbed OS Pull Request checks at the end of September 2018. Average runtime of unit tests in CI is 4.42 minutes. Test result feedback times were faster for Mbed OS PRs than previously when only Greentea tests were run on IoT hardware. Runtime for building and running all the unit tests using Mbed OS unit testing framework is currently 118.1 seconds or less than two minutes which means the unit test job in CI adds several minutes to the overall runtime.

Table 6. Mbed OS test tool performance for same unit test

Test	Test case runtime (s)	Total runtime (s)
Unit test (tool directly)	0.01	0.11
Unit test (Mbed CLI)	0.01	0.59
Greentea unit test (hardware)	0.06	13.3

4.3. Unit tests

The unit testing framework was introduced in Mbed OS 5.10.0. It included 33 unit tests as an example. 5.10.1 patch release migrated all existing unit tests written using CppUTest under this framework. Mbed OS unit test volume over time can be seen in the Figure 13.

Each test suite contains unit tests for a single Mbed OS source file. All test suites contain at least one test case. Figure 14 shows which modules have test suites written for. The unit testing framework allows generating test coverage reports from the test results. Currently Mbed OS unit test coverage is for 59.9% for lines of code and 42.1% for conditional or branch coverage. The coverage is calculated only for files which have tests written for so the total coverage is

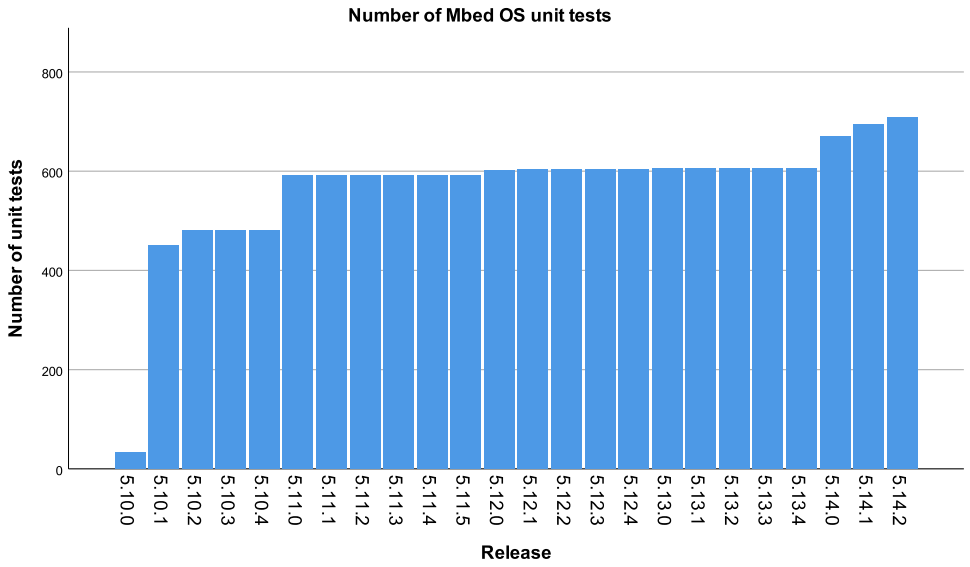

Figure 13. Number of Mbed OS unit tests

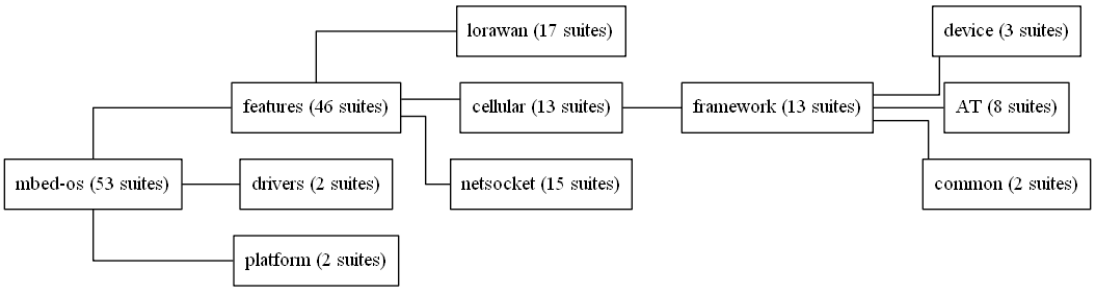

Figure 14. Unit test suite composition in Mbed OS

less than that. A detailed summary of the code coverage results is presented in Appendix 2.

4.4. Effectiveness analysis

Mbed OS unit test framework evaluation using testing tool metrics listed in Chapter 2 can be seen in the Table 7. Human Interface Design was calculated by adding KMS which is zero for the framework since only the keyboard is used, IFPF which is one for a single command line input field, and ALIF which is 35 for average test command length together. BR value was ignored since there are no buttons on this user interface. Maturity was calculated by adding M which is one year for this tool, CB which is one customer (Arm), and P which is zero since this is the first time the framework is used in any project together. Tool management was calculated by adding AL and ICM together which both are zero for this framework. Test case generation was calculated by adding ATG which is zero and TRF which is 8 since test cases are modifiable together. Feature

support was calculated as 3 since the framework is extensible, is integrated with development tools and provides test reports.

Human Interface Design score is high because the unit testing tool is only a command-line interface and there is no graphical user interface available. Maturity is quite low because the tool is used only for one year in Mbed OS testing. Tool Management is good because the framework consists of only open source software and it is freely customizable. Test Case Generation score is average because while tests have to written manually (file generation from a template is available) tests can be freely updated. Feature Support value is good. The framework can be further extended and it is integrated into Mbed CLI development tools. The framework also provides test reports and coverage reports. The rest of the metrics can be harder to calculate with limited amount of information of the project so they are left out.

Table 7. Mbed OS unit testing framework effectiveness

Metric	Value	Verdict
Human Interface Design	36	Command-line interface
Maturity & Customer Base	2	Used only on Mbed OS testing
Tool Management	0	Open to use by everyone
Test Case Generation	8	Tests written manually
Feature Support	3	Extendable and integrated Mbed CLI

5. DISCUSSION

Analysis of state-of-the-art open-source IoT operating systems showed that all operating systems have tests available. Functional tests are the most popular tests in these projects. Test applications are built for the target platform and then ran inside the device. Unit tests were also popular for quick testing. Most operating system projects had unit tests which could be run on the development platform or at least in the hardware.

All projects have Continuous Integration in place for validating incoming Pull Requests. Common checks for all CIs are building tests, building documentation, coding style analysis and static code analysis. Running unit tests is also popular in most CIs. Hardware testing in PRs is done in a few of these projects.

Mbed OS unit testing framework solved previous issues in Mbed OS testing. It allows developers to quickly validate written code without running integration tests on real-life IoT devices. Developers can compile tests to x86 architecture and to their native development platforms using native build tools. The framework provides a cross-platform command-line interface which simplifies the process of writing, building and running unit tests.

The framework is used in Mbed OS CI to run unit tests for Pull Requests and nightly tests. Unit tests are the first test feedback which developers can get after creating a Pull Request. Unit tests are run as a native process on PC platforms allowing faster testing than running those tests on the actual IoT hardware which results showed.

6. CONCLUSION

The goal of this thesis was to analyze and improve the testing methods of open source embedded operating systems for IoT. Mbed OS testing methods were improved by implementing a state-of-the-art Mbed OS unit testing framework. The framework is now used by Mbed OS developers and over 700 unit tests have been written since the tool has been released. Unit tests are also integrated into Mbed OS Continuous Integration for validating Github Pull Requests.

6.1. Achievements

A unit testing framework was implemented for Mbed OS testing. Unit tests are available in Mbed OS CI which can lessen the amount of hardware testing in CI.

I was able to conduct a study of existing unit tests frameworks and choose the best underlying test framework for Mbed OS based on the tool review. I implemented a framework which utilises many different tools behind a unified user interface. This allows developers to use this tool in their daily development and testing process.

6.2. Limitations

Comparing the testing methods of different operating system projects is difficult because all the projects are different and have different sets of features which need to be tested. Quality of tests is not considered since it depends on the testable component. The focus of this thesis is to only analyse the testing methods of these operating systems and try to find common methods which can be implemented on open source embedded operating system projects for IoT.

This thesis only analyses the testing methods of six state-of-the-art open source embedded operating systems for low-end IoT devices. These operating systems were chosen because they all target similar MCUs and architectures such as Cortex-M-family. More in depth research is needed to find more ways to improve the testing methods of IoT operating systems.

Measuring how Mbed OS CI was improved by unit tests alone is difficult because CI was constantly changing during that time and will not give reliable results. It was also difficult to change the testing process and get developers to use the framework. Some developers were more comfortable with writing hardware unit tests using Greentea than using the unit testing framework.

6.3. Future Work

The unit testing framework is limited only on testing internal operating system modules. Mbed OS unit testing framework could be integrated into Mbed OS tools for allowing application developers to use the framework to test their own components. This would require changing how the tool resolves file paths to test

sources and code coverage reports. The tool is very generic and can easily be used in any C/C++ software project.

The framework requires to use Gcovr for generating code coverage reports. The framework could be made more modular to allow using other tools for code coverage report generation.

More research can be done to measure how efficient the testing tools of different IoT operating systems are. Improving testing on a simulator is also an interesting topic which would need to further analyse different operating system projects. Artificial intelligence and machine learning could also be used to improve testing methods. State-of-the-art static analysis tools can scan code and detect problems but current operating system testing tools still lack automation and machine learning especially for generating new tests.

7. REFERENCES

[1] Popular internet of things forecast of 50 billion devices by 2020 is outdated. URL: https://spectrum.ieee.org/tech-talk/telecom/internet/popular-internet-of-things-forecast-of-50-billion-devices-by-2020-is-outdated, accessed: 24.11.2019.

[2] Who needs the internet of things? URL: https://www.linux.com/news/who-needs-internet-things, accessed: 24.11.2019.

[3] Atzori L., Iera A. & Morabito G. (2010) The internet of things: A survey. Computer Networks , pp. 2787–2805.

[4] Why a real-time operating system is a necessity for iot. URL: https://www.intervalzero.com/rtos/real-time-operating-system-necessity-iot/, accessed: 08.12.2019.

[5] Myers G.J., Sandler C. & Badgett T. (2011) The art of software testing. John Wiley & Sons.

[6] Cohn M. (2010) Succeeding with agile: software development using Scrum. Pearson Education.

[7] Fowler M., The practical test pyramid. URL: https://martinfowler.com/articles/practical-test-pyramid.html, accessed: 28.11.2019.

[8] Iot testing tutorial: What is, process, challenges & tools. URL: https://www.guru99.com/iot-testing-challenges-tools.html, accessed: 24.11.2019.

[9] Simulated vs. real-device testing. URL: https://saucelabs.com/blog/simulated-vs-real-device-testing, accessed: 01.12.2019.

[10] Huizinga D. & Kolawa A. (2007) Automated Defect Prevention : Best Practices in Software Management. John Wiley & Sons.

[11] Top continuous integration tools: The 50 best ci & continuous delivery tools. URL: https://stackify.com/top-continuous-integration-tools/, accessed: 01.12.2019.

[12] Michael J.B., Bossuyt B.J. & Snyder B.B. (2002) Metrics for measuring the effectiveness of software-testing tools. In: 13th International Symposium on Software Reliability Engineering, 2002. Proceedings., pp. 117–128.

[13] Qutqut M.H., Al-Sakran A., Almasalha F. & Hassanein H.S. (2018) Comprehensive survey of the iot open-source oss. IET Wireless Sensor Systems 8, pp. 323–339.

[14] Apache mynewt. URL: https://mynewt.apache.org/, accessed: 30.11.2019.

[15] Ng: The os for next generation iot devices. URL: http://contiki-ng.org/, accessed: 30.11.2019.

[16] Market leading rtos (real time operating system) for embedded systems with internet of things extensions. URL: https://freertos.org/, accessed: 30.11.2019.

[17] (2012), Freertos. URL: https://aws.amazon.com/freertos/.

[18] Verifying a freertos port? URL: https://www.freertos.org/FreeRTOS_Support_Forum_Archive/June_2012/freertos_Verifying_a_FreeRTOS_port_5375606.html, accessed: 30.11.2019.

[19] Mbed os 5. URL: https://os.mbed.com/, accessed: 30.11.2019.

[20] Certified product: Mbed os 5.12 (arm mbed os). URL: https://www.psacertified.org/products/mbed-os/, accessed: 30.11.2019.

[21] Testing. URL: https://os.mbed.com/docs/mbed-os/v5.14/tools/testing.html, accessed: 30.11.2019.

[22] Klasila A. (2019) Mbed OS regression test selection and optimization. Master's thesis, University of Oulu.

[23] The friendly operating system for the internet of things. learn more. URL: https://riot-os.org/, accessed: 30.11.2019.

[24] Will H., Schleiser K. & Schiller J. (2009) A real-time kernel for wireless sensor networks employed in rescue scenarios. In: 2009 IEEE 34th Conference on Local Computer Networks, pp. 834–841.

[25] Embedded unit. URL: https://sourceforge.net/p/embunit/wiki/Home/, accessed: 30.11.2019.

[26] kaspar030, kaspar030/murdock. URL: https://github.com/kaspar030/murdock, accessed: 30.11.2019.

[27] Home. URL: https://www.zephyrproject.org/, accessed: 30.11.2019.

[28] Firesmith D. (2013), Common testing problems: Pitfalls to prevent and mitigate. URL: https://insights.sei.cmu.edu/sei_blog/2013/04/common-testing-problems-pitfalls-to-prevent-and-mitigate.html.

8. APPENDICES

Appendix 1 Unit testing tool review summary
Appendix 2 Mbed OS unit test coverage report

	Google Test	CppUTest	Catch2	Boost.Test	Unity	CppUnit
xUnit	Yes	Yes	No	Yes	Yes	Yes
Header-only	No	No	Yes	Variant	No	No
Discrete test names	Yes	Yes	Yes	Yes	Only by function name	Yes
Auto-registering tests	Yes	Yes	Yes	Yes	W/ test runner or Ceedling	Yes (suites)
Fixtures	Yes	Yes	Yes	Yes	W/ Add-on	Yes
Process isolated tests	No	No	No	No	No	No
Parallel run	W/ test runner		W/ test runner			
Thread safety	W/ pthreads	No	No	No	No	Yes
Equality assertions	Yes	Yes	No (require, check)	Yes	Yes	Yes
Matchers	Yes	No	Yes	No	No	
Test suites	Yes	Yes	Yes	Yes	No	Yes
Data generators / Parameterized tests	Yes (value-param. And type-param.)	No	Yes, data generator only	Yes (data driven)	Yes, data generator only	No
Mocks	Yes (gmock)	Yes (CppUMock)	No	W/ Turtle	W/ Cmock	No
Exceptions	Yes	No	Yes	Yes	W/ Cexception	Yes
Language	C++	Limited C++	C++	C++	C	C++
Std version	C++98, C++11	C++98	C++11/14/17, (Catch 1.x C++03)	C++03, C++11/14	C89, C99	C++98
Build platforms	Linux, MinGW, Mac OS X, Cygwin	Linux, Mac OS X, Cygwin	Linux, Windows, Mac OS X	Linux, Windows, Mac OS X	Linux, MinGW, Mac OS X, Cygwin	Windows, Unix (automake)
Memory leak detection	No	Yes	No	No	No	No
Test report generation	XML	XML (junit, teamcity)	XML (junit, teamcity, tap, automake), Custom	XML (junit, generic), HRF	W/ Ceedling	XML
	User-defined assertions, death tests, fatal and non-fatal failures	Limited C++ set using macros, Memory leak detection, TDD	TDD & BDD	Assertion context for advanced diagnostic, apparantly a pain to set up	TDD with Ceedling build tool, embedded development focused	

GCC Code Coverage Report

Directory: ./
Date: 2019-11-29 23:24:09
Legend: low: < 75.0 % medium: >= 75.0 % high: >= 90.0 %

	Exec	Total	Coverage
Lines:	9322	15559	59.9 %
Branches:	4341	10300	42.1 %

File	Lines			Branches	
components/storage/blockdevice/COMPONENT_DATAFLASH/DataFlashBlockDevice.cpp		0.0 %	0 / 218	0.0 %	0 / 95
components/storage/blockdevice/COMPONENT_FLASHIAP/FlashIAPBlockDevice.cpp		0.0 %	0 / 86	0.0 %	0 / 54
components/storage/blockdevice/COMPONENT_I2CEE/I2CEEBlockDevice.cpp		0.0 %	0 / 65	0.0 %	0 / 42
components/storage/blockdevice/COMPONENT_QSPIF/QSPIFBlockDevice.cpp		0.0 %	0 / 741	0.0 %	0 / 535
components/storage/blockdevice/COMPONENT_RSPIF/SPIFReducedBlockDevice.cpp		0.0 %	0 / 119	0.0 %	0 / 84
components/storage/blockdevice/COMPONENT_SD/SDBlockDevice.cpp		0.0 %	0 / 431	0.0 %	0 / 243
components/storage/blockdevice/COMPONENT_SPIF/SPIFBlockDevice.cpp		0.0 %	0 / 405	0.0 %	0 / 274
drivers/source/PwmOut.cpp		67.1 %	57 / 85	75.0 %	9 / 12
drivers/source/Watchdog.cpp		78.0 %	32 / 41	50.0 %	14 / 28
events/source/equeue.c		91.2 %	344 / 377	83.1 %	128 / 154
features/cellular/framework/AT/ATHandler.cpp		94.0 %	825 / 878	74.1 %	524 / 707
features/cellular/framework/AT/AT_CellularBase.cpp		100.0 %	14 / 14	100.0 %	4 / 4
features/cellular/framework/AT/AT_CellularContext.cpp		82.5 %	390 / 473	51.6 %	198 / 384
features/cellular/framework/AT/AT_CellularDevice.cpp		79.4 %	243 / 306	58.3 %	112 / 192
features/cellular/framework/AT/AT_CellularInformation.cpp		100.0 %	43 / 43	91.7 %	22 / 24
features/cellular/framework/AT/AT_CellularNetwork.cpp		79.1 %	273 / 345	48.1 %	150 / 312
features/cellular/framework/AT/AT_CellularSMS.cpp		73.2 %	418 / 571	49.3 %	205 / 416
features/cellular/framework/AT/AT_CellularStack.cpp		71.9 %	161 / 224	44.9 %	89 / 198
features/cellular/framework/common/CellularUtil.cpp		94.1 %	193 / 205	76.5 %	130 / 170
features/cellular/framework/device/CellularContext.cpp		92.8 %	77 / 83	61.8 %	42 / 68
features/cellular/framework/device/CellularDevice.cpp		82.9 %	102 / 123	59.5 %	50 / 84
features/cellular/framework/device/CellularStateMachine.cpp		82.1 %	330 / 402	59.7 %	142 / 238
features/device_key/source/DeviceKey.cpp		0.0 %	0 / 110	0.0 %	0 / 102
features/frameworks/nanostack-libservice/source/libBits/common_functions.c		0.0 %	0 / 32	0.0 %	0 / 16
features/frameworks/nanostack-libservice/source/libip4string/ip4tos.c		100.0 %	21 / 21	100.0 %	10 / 10
features/frameworks/nanostack-libservice/source/libip4string/stoip4.c		95.0 %	19 / 20	76.9 %	20 / 26
features/frameworks/nanostack-libservice/source/libip6string/ip6tos.c		79.5 %	35 / 44	80.0 %	16 / 20
features/frameworks/nanostack-libservice/source/libip6string/stoip6.c		64.0 %	48 / 75	60.6 %	40 / 66
features/lorawan/LoRaWANInterface.cpp		100.0 %	84 / 84	50.0 %	53 / 106
features/lorawan/LoRaWANStack.cpp		94.0 %	564 / 600	78.7 %	281 / 357
features/lorawan/lorastack/mac/LoRaMac.cpp		84.7 %	850 / 1004	52.7 %	299 / 567
features/lorawan/lorastack/mac/LoRaMacChannelPlan.cpp		100.0 %	63 / 63	100.0 %	30 / 30
features/lorawan/lorastack/mac/LoRaMacCommand.cpp		98.7 %	222 / 225	80.6 %	50 / 62
features/lorawan/lorastack/mac/LoRaMacCrypto.cpp		99.3 %	151 / 152	72.4 %	71 / 98
features/lorawan/lorastack/phy/LoRaPHY.cpp		96.5 %	538 / 561	83.3 %	280 / 336
features/lorawan/lorastack/phy/LoRaPHYAS923.cpp		100.0 %	106 / 106	66.7 %	24 / 36
features/lorawan/lorastack/phy/LoRaPHYAU915.cpp		97.3 %	218 / 224	74.5 %	76 / 102
features/lorawan/lorastack/phy/LoRaPHYCN470.cpp		96.9 %	190 / 196	67.5 %	54 / 80
features/lorawan/lorastack/phy/LoRaPHYCN779.cpp		100.0 %	66 / 66	50.0 %	1 / 2
features/lorawan/lorastack/phy/LoRaPHYEU433.cpp		100.0 %	66 / 66	50.0 %	1 / 2
features/lorawan/lorastack/phy/LoRaPHYEU868.cpp		100.0 %	71 / 71	50.0 %	1 / 2
features/lorawan/lorastack/phy/LoRaPHYIN865.cpp		100.0 %	68 / 68	50.0 %	2 / 4
features/lorawan/lorastack/phy/LoRaPHYKR920.cpp		99.3 %	145 / 146	60.5 %	46 / 76
features/lorawan/lorastack/phy/LoRaPHYUS915.cpp		99.2 %	244 / 246	72.2 %	78 / 108
features/lorawan/system/LoRaWANTimer.cpp		100.0 %	25 / 25	50.0 %	2 / 4
features/nanostack/coap-service/test/coap-service/unittest/stub/mbedtls_stub.c		58.7 %	88 / 150	33.9 %	21 / 62
features/netsocket/DTLSSocket.cpp		18.8 %	3 / 16	0.0 %	0 / 26
features/netsocket/DTLSSocketWrapper.cpp		55.6 %	15 / 27	38.9 %	7 / 18
features/netsocket/EMACInterface.cpp		60.8 %	59 / 97	42.6 %	40 / 94
features/netsocket/EthernetInterface.cpp		100.0 %	2 / 2	- %	0 / 0
features/netsocket/InternetDatagramSocket.cpp		89.4 %	76 / 85	63.4 %	52 / 82
features/netsocket/InternetSocket.cpp		95.6 %	108 / 113	64.8 %	57 / 88
features/netsocket/NetworkInterface.cpp		84.7 %	61 / 72	64.6 %	31 / 48
features/netsocket/NetworkStack.cpp		28.8 %	47 / 163	32.9 %	46 / 140
features/netsocket/SocketAddress.cpp		93.0 %	107 / 115	89.7 %	52 / 58
features/netsocket/TCPServer.cpp		100.0 %	34 / 34	60.5 %	23 / 38
features/netsocket/TCPSocket.cpp		93.3 %	152 / 163	67.2 %	86 / 128
features/netsocket/TLSSocket.cpp		25.0 %	3 / 12	0.0 %	0 / 10
features/netsocket/TLSSocketWrapper.cpp		94.5 %	274 / 290	71.3 %	154 / 216
features/netsocket/UDPSocket.cpp		100.0 %	5 / 5	50.0 %	1 / 2
features/netsocket/WiFiAccessPoint.cpp		100.0 %	16 / 16	- %	0 / 0
features/netsocket/cellular/CellularNonIPSocket.cpp		80.3 %	98 / 122	72.4 %	42 / 58
features/netsocket/nsapi_dns.cpp		82.4 %	491 / 596	54.9 %	301 / 548
features/storage/blockdevice/BufferedBlockDevice.cpp		0.0 %	0 / 165	0.0 %	0 / 122
features/storage/blockdevice/ChainingBlockDevice.cpp		0.0 %	0 / 138	0.0 %	0 / 114
features/storage/blockdevice/ExhaustibleBlockDevice.cpp		0.0 %	0 / 87	0.0 %	0 / 52
features/storage/blockdevice/FlashSimBlockDevice.cpp		0.0 %	0 / 100	0.0 %	0 / 66
features/storage/blockdevice/HeapBlockDevice.cpp		98.9 %	90 / 91	75.0 %	45 / 60
features/storage/blockdevice/MBRBlockDevice.cpp		0.0 %	0 / 199	0.0 %	0 / 138
features/storage/blockdevice/ObservingBlockDevice.cpp		0.0 %	0 / 41	0.0 %	0 / 16
features/storage/blockdevice/ProfilingBlockDevice.cpp		0.0 %	0 / 49	0.0 %	0 / 8
features/storage/blockdevice/ReadOnlyBlockDevice.cpp		0.0 %	0 / 31	0.0 %	0 / 2
features/storage/blockdevice/SlicingBlockDevice.cpp		67.2 %	41 / 61	53.1 %	17 / 32
features/storage/filesystem/Dir.cpp		0.0 %	0 / 37	0.0 %	0 / 20
features/storage/filesystem/File.cpp		0.0 %	0 / 48	0.0 %	0 / 28
features/storage/filesystem/FileSystem.cpp		0.0 %	0 / 73	0.0 %	0 / 18
features/storage/kvstore/conf/kv_config.cpp		0.0 %	0 / 209	0.0 %	0 / 230
features/storage/kvstore/direct_access_devicekey/DirectAccessDevicekey.cpp		0.0 %	0 / 87	0.0 %	0 / 58
features/storage/kvstore/filesystemstore/FileSystemStore.cpp		0.0 %	0 / 320	0.0 %	0 / 254
features/storage/kvstore/global_api/kvstore_global_api.cpp		0.0 %	0 / 107	0.0 %	0 / 80
features/storage/kvstore/kv_map/KVMap.cpp		0.0 %	0 / 158	0.0 %	0 / 90
features/storage/kvstore/tdbstore/TDBStore.cpp		0.0 %	0 / 756	0.0 %	0 / 590
features/storage/system_storage/SystemStorage.cpp		0.0 %	0 / 17	0.0 %	0 / 26
platform/source/ATCmdParser.cpp		93.4 %	156 / 167	76.7 %	112 / 146

Generated by: GCOVR (Version 4.1)