TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies

Eslam El-Sherbieny 166778IVSM/B66096

TTU SELF DRIVING CAR MASTER CONTROLLER EMBEDED SOFTWARE TESTING

Master's thesis

Supervisor: PhD Mairo Leier

Tallinn 2018

TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Eslam El-Sherbieny 166778IVSM/B66096

TTÜ ISESÕITVA AUTO JUHTKONTROLLERI TARKVARA TESTIMINE

Magistritöö

Juhendaja: PhD Mairo Leier

Tallinn 2018

Author’s declaration of originality

I hereby certify that I am the sole author of this thesis. All the used materials, references to the literature and the work of others have been referred to. This thesis has not been presented for examination anywhere else.

Author: Eslam El-Sherbieny

07.05.2018

Abstract

The objective of this thesis is to research the available embedded software testing frameworks, select the most suitable one based on the requirements and cover the existing functionality with tests.

The framework used is developed by Throwtheswitch community [1]. It is an open source unit testing framework following the Junit testing techniques.

Tests results shows the advantage of integrating such a lightweight module into the project and taking advantage of its rich set of assertions, with almost no intrusion on the original code and negligible performance overhead.

This thesis is written in English and is 48 pages long, including 6 chapters, 39 figures and 3 tables.

List of abbreviations and terms

TTU	Tallinn Technical University
PC	Personal Computer
I/O	Input/Output
GPS	Global Positioning System
LIDAR	Light Detection and Ranging
RTOS	Real Time Operating System
CAN	Controller Area Network
IDE	Integrated Development Environment
USART	Universal Synchronous and Asynchronous Receiver-Transmitter
ISR	Interrupt Service Routine
USB	Universal Serial Bus

Table of contents

List of figures

List of tables

1 Introduction

Testing is a primary factor in any project, through it, a user can check and refactor any part of the project to correct and enhance it. With testing it can be guaranteed that the final product will work as it was intended, providing a secure, safe and efficient result to the end-user.

Whether the project is in the field of engineering, medical, physics, etc. Testing is a crucial milestone that has to be thoroughly applied.

Embedded software engineering is not different when it comes to the importance of testing, the development of efficient, functional and error free software is something of a great importance in the embedded field. This is due to the critical nature of almost all the applications of embedded systems, failure means some problem is imminent, a fatal one.

In this thesis I am developing the embedded software tests of Tallinn Technical University autonomous vehicle project. There are a lot of concepts and proven methodologies out there that will be discussed in the next section, which defines how I will approach the testing phase of the project.

I focus on developing a testing functionality to perform low level master controller software tests. The master controller, responsible for command and communication between various embedded devices which are in turn responsible for controlling the vehicle's inputs (steering, velocity, brakes, etc.). Testing starts with individual testing of each code module, and gradually it should integrate testing suites to cover a broader cycle of the project, eventually it should cover the whole system.

This thesis can be divided into three chapters; In the first chapter, I look for suitable testing frameworks that would suit the software needs. The second chapter is dedicated to the installation trials of the selected frameworks into the existing software code and the validation of how well it works and how much interference it has on the processing of the existing system. In the last chapter, covers the testing phase where I create a testing suite for every testable module.

1.1 Software Testing History

In the early days of computing, back when cross-platform programming languages like C were not yet invented and programs depended heavily on assembly code that worked on only one specific type of computer chip, software was rarely designed to run in multiple environments. That made configuration testing unnecessary, since there were fewer configurations to test for [1].

Users would have almost identical computers to the developers, otherwise the software would not work. By the 1980s when IBM introduced the first PC as a commercial product, the industry evolved dramatically, by the 1990's many PC variations were on the market and high demand for cross-platform software pushed the programmers to develop their products to be compatible with any type of computer that was advertised as PC-compatible. Another change was the increasing demand for more frequent software releases, which was derived by the commercialization of PCs and the growing importance of the internet [1].

Releasing software that worked on any PC and the expectation that these software will frequently be updated to improved versions with new features in a fast pace raised the stakes for software testing. It required careful configuration and testing on many possible environment variables. [1]

1.2 Background

Testing is the process which focuses on finding defects in a system, testing is a crucial building block in system development; it works towards improving the system's quality, and avoid risks of software failures after deployment to the commercial market. [2].

In software development, we gather requirements, create high level designs, low level design, develop the code, test it, refactor, test again, and refactor until we are satisfied, then we integrate and start a final testing round before deployment. Since most projects run late, testing sometimes is sacrificed partially to mitigate the delays that happened through the project cycle, in order to deliver on time. This is a bad habit that many companies are avoiding these days [3].

Best practice development includes frequent code checks, but these only find typically 70% of the system's bugs, so a thorough testing plan is essential to every software product. In all other disciplines of engineering, testing is considered fundamental, whether it's architectural, mechanical, etc. [3].

Embedded systems software testing shares much in common with application software testing. However, some important differences exist between application testing and embedded systems testing. [3]

1.2.1 **Self-Driving Car Project**

The software to be tested is being developed in the Technical University of Tallinn, Estonia. Its aim is to produce a system for a self-steered vehicle which would allow the vehicle to safely pass through the predetermined route traffic. The vehicle has predefined parameters of speed, dimensions, engine power, etc [4].

Figure 1 The Self-driving car model [4]

The vehicle's control system would follow a set of predefined Global Positioning System (GPS) coordinates along the TUT campus available for public use, it's would respect the barriers (parked cars, pedestrians, etc.) and passes them safely [4].

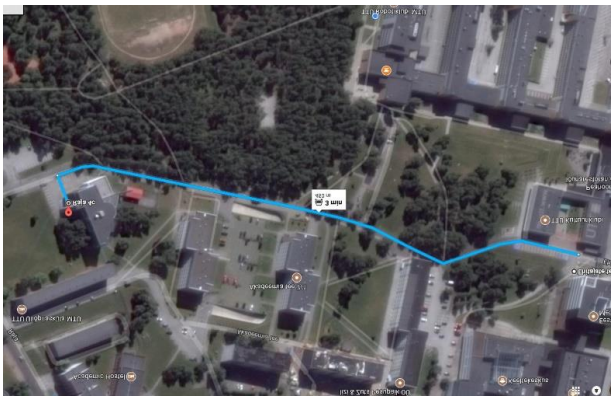

Figure 2 Predefined road [4]

The self-driving project consist of many parts that works together to achieve the main goal of the project:

- Master controller (where this thesis tests are applied to)
- Drive controller (slave controller responsible for the car actions)
- Body controller
- Several sensors, communication devices, drivers
- Car actuators, battery, wireless network module (which connects to a data center through 5G)
- PC Autoware connected to the master controller and to other sensors through Ethernet.

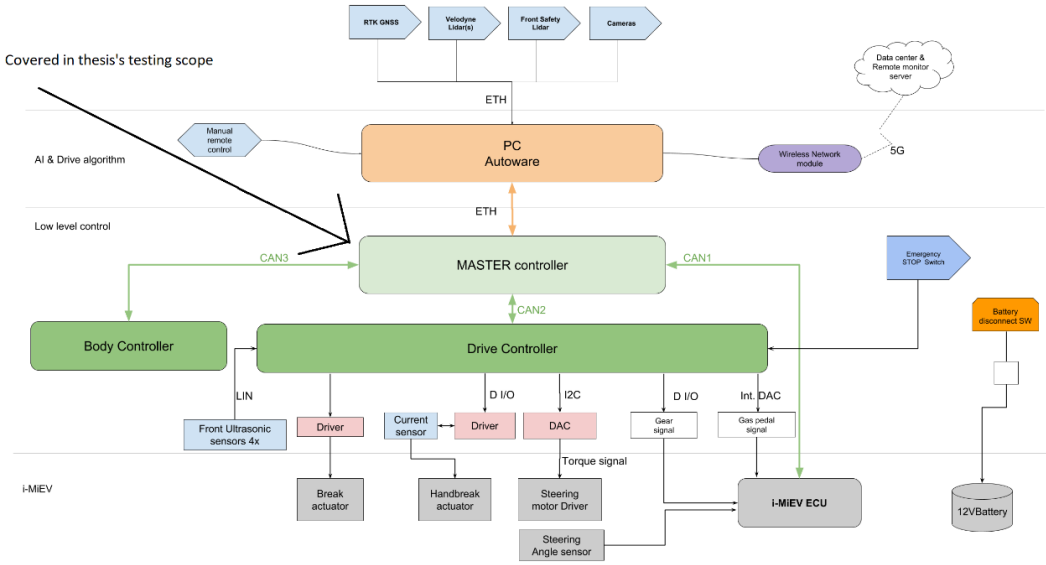

Figure 3 Vehicle hardware diagram [4]

Given the sensitivity and criticality of the project's application, it was crucial to apply testing to the code being developed for this project. It was important to guarantee that the

system is behaving as per design and within the safety regulations required in the automotive industry.

1.2.2 **Functional Testing**

It is the testing approach that focuses on the software's validity with respect to the requirements. Checking that each function and module operates as per the intended design of the program. It consists of three different variations, black-box testing, white-box testing and grey-box testing [3].

- Black-box testing focuses on testing the functional behaviour of the code, without knowing the actual code. It depends on testing the Input/Output (I/O) of the function, program and/or the device.
- White-box testing focuses on testing the reliability and correctness of the code itself, checking every functions logic, execution flow, statement coverage, decision coverage if conditional logic is found.
- Grey-box testing is a mix of both of the above types, it gives the tester some space to intrude the function's logic but not stressing too much on his knowledge of the code.

Functional testing follows the requirements and specifications provided by the management which were used by the design and implementation teams when developing the program.

1.2.3 **Non-Functional Testing**

This type of testing focuses on the performance, usability and reliability of the program among other things. It tests the non-functional requirements compiled by the management of the project, which are not addressed in the functional testing.

- Stress tests: putting the system limits to the test and checking the response, overloading inputs and memory [5].
- Scalability tests: testing the ability to increase the defined limits of any of the non-functional requirements [5].
- Compatibility tests: testing the software's ability to coordinate with different hardware and software that it should work with [5].
- Usability tests: tests that verifies that the software is user friendly and easy to use.

Functional Testing	**Non-Functional Testing**
Cover business requirements	Covers performance requirements
Executed first	Executed later after functional testing is done
Tests how the program is behaving	Tests how the program is performing
Handles customer requirements	Handles customer expectations
Types includes: Unit testing, Integration testing, Smoke/Sanity, User Acceptance, Localization, Globalization, Interoperability, etc.	Types includes: Performance, Endurance, Load, Volume, Scalability, Usability, etc.

Table 1 Testing Methodology Comparison

In this thesis functional testing is used, focusing on the functions' behavior in each module. I used a mix of both Black and White box testing wherever one was more suitable than the other, depending on the function's logic.

1.2.4 **Software Types**

Knowing how the software typically fails should influence how to select the tests, since embedded systems depend heavily on asynchronous events, which are by nature unpredictable, the tests should cover failures that can exist from these kinds of events. In every real-time system, a sequence of events, would cause a great delay from the event trigger to its response. The embedded test suite should be capable of generating all these sequences and measuring the associated response time [6].

- Embedded software must be reliable to run for a long time without problems [6].
- Embedded software is often used in critical applications that involves human lives [6].
- Embedded systems always work in resource constrained environments, which gives no chance for the software to be inefficient [6].
- Embedded software must act as a problem solver to mitigate hardware faults [6].
- Real-world events are usually asynchronous and nondeterministic; therefore, no simulation tests can be depended on [6].
- Failure probably means disaster or crisis.

- Embedded developers often have access to hardware-based tools for testing that are generally not used in application development [7].
- Most embedded systems are resource-constrained real-time systems, more performance and capacity testing are required [6].
- Some real-time trace tools can be used to measure how well the tests are covering the code [8].
- Testing need to be aiming for a higher level of reliability than if you were testing application software. [6]

2 Research

In order to approach the task in a correct way, detailed analysis of the system, the devices involved, the running operating system, programming languages, communication protocols, and the functional objectives of the software is gathered and studied.

The main microcontroller is at the vehicle's core, other microcontrollers are integrated, multiple sensors, ranging from multiple ultrasonic to one primary Light Detection and Ranging (LIDAR) sensor. All microcontrollers are running freeRTOS [9] a real time operating system, the core programming languages are C and C++. Internal connections and communication are mostly in Controller Area Network (CAN) and some in Ethernet. As for the functional objectives: autonomous steering, braking, cruise control, object detection, GPS localization, velocity and odometry measurement, live data streaming to servers for analysis and observation, etc.

The software code skeleton is built using STM32CubeMx code generator with specifications decided by the design and implementation teams. The development is done in System Workbench; an Integrated Development Environment (IDE) based on Eclipse made by OpenSTM32 community [10]. This thesis scope is the parts developed by the teams and not the generated code.

2.1 System Requirements

First phase of the project is to find a suitable testing framework, many exists on the market already with different properties and variations. Due to the nature of the system I am targeting, certain preferences are taken into account when considering candidates of frameworks.

- The software is being developed in an embedded environment, which means that the resources would be scarce.
- Power, time and processing consumption are to be minimized.
- Support for Real Time Operating System (RTOS) tasks and interrupts.
- Testing automation support.
- Performance testing.

- Compatible with Windows operating system as the implementation teams are using it as the development system.
- Free if possible.

2.2 Testing Framework Selection Criteria

With the above factors in mind, a number of requirements that would help in choosing the correct framework for the software is compiled. Online search for all the available testing frameworks for C and C++ and cross checking the properties of each with the pre-defined requirements is done.

Title	Supported languages	Embedded Focused	RTOSsupport	Performance test support	Test automation support	Resource Constrained support	OS support	Cost	Preparations needed to support Keil
AceUnit	C	Yes	No	No	Yes	Yes	Windows	Free	not available
GNU Autounit	C, Guile	No	No	No	Yes	No	No	Free	not available
CUnit	C	No	No	No	Yes	No	Windows	Free	not available
CuTest	C	No	No	No	No	No	Windows	Free	not available
CppUnit	C, C++	No	Yes	Yes	Yes	No	Windows	Free	not available
embUnit	C	Yes	Yes	No	Yes	Yes	Windows	Free	not available
MinUnit	C	Yes	No	No	No	Yes	Windows	Free	not available
CMocka	C	Yes	No	No	No	No	Windows	Free	not available
Criterion	C, C++	No	No	No	Yes	No	Windows	Free	not available
HWUT	C and others	No	No	No	Yes	No	Windows	Free	not available
Google Test Framew	C, C++	No	No	Yes	Yes	Yes	Windows	Free	not available
Unity									
CMock									
Ceedling									
CException	C	Yes	No	No	Yes	Yes	Windows	Free	Tutorial
CMockery	C	No	No	No	Yes	Yes	Windows	Free	not available
Cheat	C	Yes	No	No	Yes	Yes	Windows	Free	not available
Seatest	C	No	No	No	Yes	Yes	Windows	Free	not available
unit	C, C++	No	No	Yes	Yes	No	Windows	Free	not available
Parasoft	C, C++	Yes	No	Yes	Yes	Yes	Windows	No	Supported
greatest	C	No	No	No	Yes	No	Windows	Free	not available
VisualGDB	C, C++	Yes	No	Yes	Yes	Yes	Windows	No	Supported

Figure 4 Testing Frameworks Requirements

During my research 18 potential frameworks were found, some of could not be tested, either the developers gave incomplete documentation regarding the features and setup, the framework was not compatible with windows, or it was not for free.

2.3 Integration Trials

Most of the frameworks found were tried with the project to some degree, for each, I downloaded the program in the available format (source code, executable, etc.) placed it within the project as a new module and started trying to build, compile and run the software to see if it will work, how good can it be utilized in the code and how much will it be affecting the performance.

- AceUnit, EmbUnit, CHEAT, Seatest
    - Small, frameworks developed in C and focusing on Embedded testing, in JUnit4.x style, they were integrated successfully with program. The negative aspect of these frameworks was the scarcity of built in assertions. I had to compose my own if I had to proceed.
- GNU Autounit
    - A small, framework for testing C and other languages, it was not made for Embedded projects, it lacked the documentation and it was depending on using GNU Autoconf [11] in the projects to be tested. Therefore, I did not go through with the integration trials
- CUnit, CuTest, HWUT, µnit, greatest
    - Those frameworks shared similar characteristics, lightweight, they are built as a static library or just a header file and linked with the testing code that would be developed, they were successfully integrated into the project but proved to be needing a lot of work to setup tests, write a test suite with and some lacked the automation feature
- Criterion, CMocka
    - Very simple frameworks for testing C, they were built and compiled with the project with no issues, but they lacked the ability to output their results through Universal Synchronous and Asynchronous Receiver-Transmitter (USART) serial port as the project is run, test suites had to be run independently from terminal to get the results out, which proved to be impractical.
- CppUnit, Google Test
    - Frameworks that are built in C++ proved to be troublesome when compiling with C based projects and C based compiler configurations. I discarded those options due to the large number of errors produced when compiling.
- Parasoft, VisualGDB
    - These frameworks promised a much better option for unit testing, they had large libraries of unit tests and support from their developing companies, but they were not for free.
- Unity, Cmockery
    - Both were lightweight, and worked with the software with no issues.

I decided to work with Unity testing framework, an open source framework built by Throwtheswitch community [12]. It proved to be lightweight, portable (which was the same case as Cmockery) and yet very rich in assertions, and it supported centralized testing automation and reports with results summary and clear success/failure identifiers that were easily hooked up with USART serial port and rendered the results back into the terminal while the project ran.

3 Integration

In order to get actual live readings from CAN and Ethernet connections in the car that is being tested in the testing environment of the project, the tests need to read values from interrupt states and incoming data streams that is only possible to read on the microcontroller itself, therefore Unity source code is installed into the project as one of its modules. Even if during these tests I have no access to the live system, it is better to prepare the environment for live testing so in future it can be easily integrated and used.

3.1 Project Structure

The software skeleton is generated using STM32CubeMx with target platform System Workbench. A new folder is created in the project for Unity, which has the source and include subfolders for its files. A Tests folder is created with source and include subfolders to include sources and headers for each module in the project, Unity is downloaded from their project's Git repository [13]. In Tests folder I first created a test runner which included a source and header that would act as the tests automation tool, calling tests at the desired time.

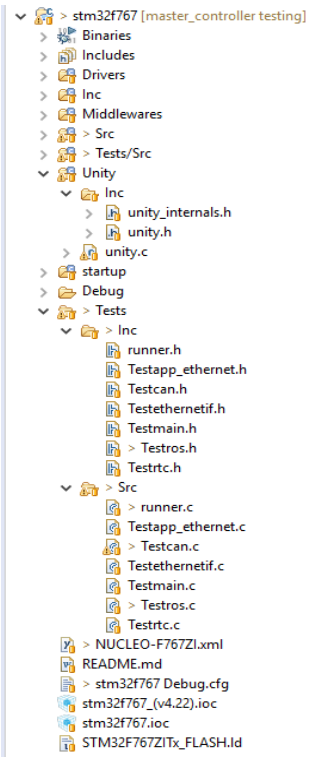

Figure 5 Software Structure

For each module, a test suite is created, that would cover each function in the module (that can be tested), in each function for clarity reasons, every variable to be tested is given a separate testing function of its own. This is because Unity stops execution of a single test function upon the first failure and moves on to the next function. So to clarify which part of the function failed in test, it proved beneficial to create the tests as:

- If a function manipulates several variables, a test for each variable in that structure is created.
- If a function has a return value, a separate test would cover it.

The Project consist of the main source folder where the program main logic resides, in each module, there are two types of code, the generated code, which is not tested and the user code which is created by the implementation teams in the project and this is where the code to be tested resides. Various other folders in the project consist of drivers, helpers, libraries and many other modules that are related to the hardware, the CubeMx generator, etc.

3.2 Tests Structure

A main test file (runner) is created to consolidate all tests runner into one initialization point, all other test related helpers also resided in this file, this approach simplifies tracing all the tests starting points and separates the tests module from the rest of the software. The source file runner.c includes all the headers of the test files created for each module, the FreeRTOS, tasks headers for creating semaphores and tasks that are needed when testing functions that work in the Interrupt Service Routine (ISR) and of course the Unity header itself.

A runner header file is included in the software main file to initialize all tests' variables and run the test suites at the desired point in the process. A test source and header with the naming convention Test[filename] (where filename is the name of source file name being tested) are created for each module that would be tested, the header of that module would be included in the test source to access the needed functions and variables that can be accessed and tested.

3.2.1 **Test Runner**

A single-entry point for all tests, this is where the tests are called to run and return their results to the output print stream where they are read and logged. The file contains a main void function with void parameter, called runUnityTests() which starts with the Unity macro UNITY_BEGIN() that is responsible for initializing Unity's process and get things ready for the incoming test calls which are made with the macro RUN_TEST(test_name) and takes one parameter which is the test function name that is to be run. At the end the test runner is stopped by calling the macro UNITY_END() which collect the previous test results, calculate the totals and print out a summary of successes, failures and ignores.

This approach proved to be working fine with normal functions, but for the ones in the ISR, a more invasive approach was needed to be able to access the functions and variables at the right moment when they are called into action. For that a freeRTOS semaphore dedicated to the tests and a task is created, that would be called whenever the test semaphore is given from within an ISR function, in order to run the tests and do the needed calculations and asserts outside of the ISR domain to avoid further disruption to the original software's functionality.

A global data structure is created in the runner and included wherever it is needed to capture some values from an ISR function and test it outside in the test suite, the needed variables would be assigned to one or more of the structure variables and read in the suite and asserted that the correct values are manipulated in the intended way.

3.2.2 **Test Task**

For ISR based functions, A task runner function in runner.c is created and inside the test semaphore test_sem_handle is called and once it is woken, the needed ISR tests are called using RUN_TEST macro, for a more organized way the test calls are wrapped in a UNITY_BEGIN() and UNITY_END() to separate these tests from the normal tests and have their own summary or results.

The test task along with the test semaphore are initialized through an init() function in runner.c and called at the beginning of the code from the main class.

3.2.3 **Test Data Structure**

A structure of arrays is created in the runner header file and initialized in its source, it contains multiple arrays of different data types to hold various values and transfer them from functions in the project to the test suites for assertion. It is initialized also from the main class at the beginning of the code before the scheduler starts, if needed, the data in it is cleared before using it in another function or module.

3.2.4 **Test Suites**

For each module in the project, a test suite is created (a test source and header files), the source file includes the target module header file in the suite, Unity, runner and other files as per the testing needs. One test suite had to have a startUp(void) and tearDown(void) functions because it is required by Unity to have at least one of each in the program, adding more of those helpers proved to invoke multiple definition errors by the compilers, so it is removed and aliases are used instead in other suites since they are optional anyway. These functions originally were intended to be used as helpers where startUp is called at the beginning of the test suite and tearDown at the end of each. I managed to mimic the behaviour in a more manual fashion in other suites where it is implicitly called at the beginning of a test function in a suite, if its needed.

Each test suite contains multiple test void functions and in each one a single assertion is called for a single variable or return value in the project.

3.3 Hardware

For the testing purposes, I was provided with a microcontroller board of the same model that is being used in the development of low level controller functionality. The board is connected to a host PC and the project along with the test modules are compiled and flashed onto the board for live testing.

3.3.1 **The Board**

The STM32F767ZIT6-Nucleo-144 manufactured by STMicroelectronics [14]. It features an ARM© Cortex©-M7 core at 216MHz, a 2 Mbytes of flash memory, Embedded ST-

LINK/V2-1 debugger/programmer which I use to connect to the board through a micro-b Universal Serial Bus (USB) port on it.

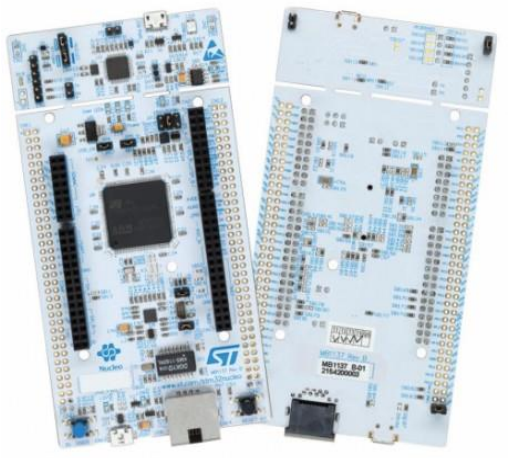

Figure 6 STM32F767 Board [15]

3.3.2 **Debug Logging Connection**

In order to get a logging mechanism working to print out the testing results from within the board, one way is to utilize the USART port communication of the board and by using a USART to USB converted I am able to get the results back to the host PC through USB and read them using a Serial port enabled terminal called TeraTerm© [16]. As shown in Figure 7 connections were made from the board to the USB converter as follows:

Board	**USART**
PA1	TXD
PA4	RXD
PC0	GND

Table 2 Board to USART converter pinout

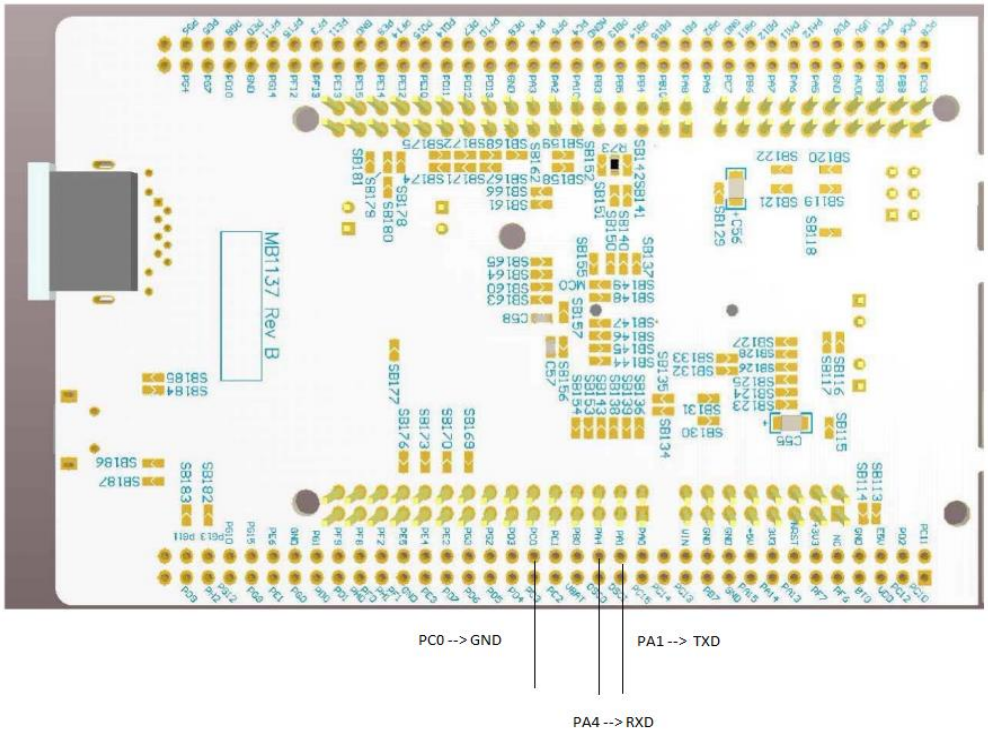

Figure 7 Board diagram (back-side) with debug connections [17]

3.4 Software Configuration

In order to correctly run the tests within the project, some minor configuration setup is done, the Unity and Tests sources and headers locations are included in the software paths on the IDE. The board USART was used for serial debugging, a serial to USB converter was connected to USART ports on the board and connected to the PC, then a TeraTerm© [16] terminal was configured with baud rate of 115200 to match the board's configuration.

3.5 Implementation

The project's code structure follows a similar pattern in all the modules that are generated by STM32CubeMx, through each module, in each section and function there are two types of code, the generated code and the user code, the former is not identified but treated as the main code in the file, the latter is identified by a comment

/ USER CODE BEGIN (keyword here) */*

User Code Here

/ USER CODE END(keyword here) */*

In place of *keyword* above there are various types clarifying which user code should go in this section, there are sections for includes, private variables, private function prototypes, and user code which are presented in a numbering convention (e.g. */* USER CODE BEGIN 0 */*) in each function there is a section that contain these comments and there a user should add his/her code.

This approach which is automatically provided by STM32CubeMX software gave the code a more organized and readable style and it is mandatory to use these regions in order to keep the code in place if the implementation team needs to change some configuration and regenerate the code again, any code inside those comments regions is not touched, this provides very high flexibility and portability throughout the project. Of course if a user creates a custom module from scratch, it also won't be touched when regenerating.

I am concerned with only the user made code, the generated code testing is out of scope of this thesis.

4 Testing

The project consists of different types of functions; initialization functions that creates a data structure or something similar, functions with parameters, and ISR functions. Mostly all the functionality is handled by the RTOS scheduler. For each type of functions, a different testing approach is needed, some are simply passing data or saving it into a data variable, so all that is needed to do is capture that variable and test it after or during the function call, some have more complex nature whether it's an ISR which prevents us from heavily intruding the code, functions that depends on incoming data streams are very difficult to test without the live environment.

In almost all test cases, they are called from the main module after all the initializations have been done and just before the scheduler takes over, this is done by called a simple function *runUnityTests()* and it then starts executing the test suites automatically.

```
void runUnityTests(void){

    printf(
            "-------------------------UNITY TESTING START-------------------------\n\n");
    //UnityBegin("Testmain.c");
    UNITY_BEGIN();

    printf("-----START || TESTING: main.c ----------\n");
    RUN_TEST(test_MPU_Config_RNR);
    RUN_TEST(test_MPU_Config_RBAR);
    RUN_TEST(test_MPU_Config_RASR);
    RUN_TEST(test_CPU_CACHE_Enable_SCB_EnableICache_ICIALLU);
    RUN_TEST(test_CPU_CACHE_Enable_SCB_EnableICache_CCR);
    RUN_TEST(test_CPU_CACHE_Enable_SCB_EnableDCache_CSSELR);
    RUN_TEST(test_CPU_CACHE_Enable_SCB_EnableDCache_CCR);
    printf("-----END || TESTING: main.c ----------\n\n");

    printf("-----START || TESTING: app_ethernet.c ----------\n");
    RUN_TEST(test_User_notification_LD1_SET);
    RUN_TEST(test_User_notification_LD1_RESET);
    RUN_TEST(test_User_notification_LD3_SET);
    RUN_TEST(test_User_notification_LD3_RESET);
    printf("-----END || TESTING: app_ethernet.c ----------\n\n");
```

Figure 8 Unit Testing Module Main Function

4.1 Initialization Functions Tests

Functions that are called early in the execution, they initialize structures and other data types responsible for transferring and/or saving data for later usage. In these functions, either the same data is checked in the test suite after it is initialized or another variable

with the same datatype is created and initialized, then the values are tested. The actual function that is to be tested is shown in the figure below, it consists of a two data structures that are initialized with values, in Figure 9, an example of testing one of those assigned values after the original function is called. It is possible to access the structures in the core code from the testing module, so no mocking is needed in that case.

```
void CAN1_Config_Filters(void) {
    CAN_FilterConfTypeDef sFilterConfig;
    hcan1.Instance = CAN1;

    //##-2- Configure the CAN Filter ###########################################
  sFilterConfig.FilterNumber = 0;
  sFilterConfig.FilterMode = CAN_FILTERMODE_IDMASK;
  sFilterConfig.FilterScale = CAN_FILTERSCALE_32BIT;
  sFilterConfig.FilterIdHigh = 0x0000;
  sFilterConfig.FilterIdLow = 0x0000;
  sFilterConfig.FilterMaskIdHigh = 0x0000;
  sFilterConfig.FilterMaskIdLow = 0x0000;
  sFilterConfig.FilterFIFOAssignment = 0;
  sFilterConfig.FilterActivation = ENABLE;
  sFilterConfig.BankNumber = 0;

  if(HAL_CAN_ConfigFilter(&hcan1, &sFilterConfig) != HAL_OK) {
    // Filter configuration Error
    Error_Handler();
  }

  //##-3- Configure Transmission process #####################################
  hcan1.pTxMsg->StdId = 0x321;
  hcan1.pTxMsg->ExtId = 0x01;
  hcan1.pTxMsg->RTR = CAN_RTR_DATA;
  hcan1.pTxMsg->IDE = CAN_ID_STD;
  hcan1.pTxMsg->DLC = 8;
}
```

Figure 9 Initialization Function with global access [18]

```
void test_CAN1_Config_Filters_StdId(void) {
    TEST_ASSERT_EQUAL_UINT32(0x321, hcan1.pTxMsg->StdId);
}

/**
 * Test: CAN1_Config_Filters_ExtId
 * Target method: CAN1_Config_Filters
 * Parent method: MievCAN2Thread
 *
 * @brief Testing the assigned values in hcan1.pTxMsg struct after calling CAN1_Config_Filters
 * Actual Value: hcan1.pTxMsg->ExtId
 * Expected Value: 0X01
 *
 */
void test_CAN1_Config_Filters_ExtId(void) {
    TEST_ASSERT_EQUAL_UINT32(0X01, hcan1.pTxMsg->ExtId);
}
```

Figure 10 Initialization Function Tests

In other cases, I have to create my own data similar to the core ones in order to test them, since there is no global access to those variables as shown in the following figures, external variables with same data types and names are created in the test suite and the initialization function called one time at the execution of the first test case.

```
void Construct_ROS_Packet(void) {
    extern struct UDPData MToROSData;
    extern struct CAN2Data iMievData;
    extern struct CAN2DataTime iMievTime;

    MToROSData.ready_can_id = CAN_ID_READY;
    MToROSData.ready_time   = iMievTime.ready;
    MToROSData.ready_data   = iMievData.ready;

    MToROSData.key_can_id = CAN_ID_KEY;
    MToROSData.key_time     = iMievTime.key;
    MToROSData.key_data     = iMievData.key;
```

Figure 11 Initialization Function without global access [18]

```
extern struct UDPData MToROSData;
extern struct CAN2Data iMievData;
extern struct CAN2DataTime iMievTime;

void ros_startUp(void) {
    Construct_ROS_Packet();
}
```

Figure 12 Initialization of the function in test suite

```
void test_Construct_ROS_Packet_can_id(void) {
    // Calling startup one time to initiate the function call
    ros_startUp();
    TEST_ASSERT_EQUAL_UINT32(CAN_ID_READY, MToROSData.ready_can_id);
}

/**
 * Test: Construct_ROS_Pack_ready_time
 * Target method: Construct_ROS_Packet
 * Parent method: UDPThread
 *
 * @brief Testing the assigned values in MToROSData struct after calling Construct_ROS_Packet
     * Actual ValueMToROSData.ready_timele
     * Expected ValueiMievTime.readyle
 *
 */
void test_Construct_ROS_Pack_ready_time(void) {
    TEST_ASSERT_EQUAL_UINT32(iMievTime.ready, MToROSData.ready_time);
}
```

Figure 13 Initialization Function Test

4.2 Function Arguments Mock Testing

Functions that take input arguments when called and do some manipulation on the data or transfer it further in the execution cycle, these functions needs to have mocks to copy the same behavior of the function as if it has actual inputs when called in the test module, since there is no way of knowing the exact values that would be coming to some of the functions from my testing environment, I decided as a temporary solution to copy the values of both the assignee and assigned and test them separately in the test module.

In some cases, (Figure 14) the values are simple, just one value assigned to a variable and tested in the suite (Figure 15)

```
// Find safety brake status
idType = MSG_TYPE_SAFETY_BRAKE;
idPosition = 10;
if (memcmp(&buf[idPosition], &idType, 1) == 0) {
    odometry.safety_brake = buf[idPosition+1];

    /**
     * Testing Area
     */

    testData.uint8_value[0] = buf[idPosition + 1];
    testData.uint8_value[1] = 135;
}
```

Figure 14 Function with Arguments 1 [18]

```
/**
 * Test: handle_odometry_packet_safety_brake
 * Target method: handle_odometry_packet
 * Parent method: udp_serve
 *
 * @brief Testing the assigned values in odometry struct after calling handle_odometry_packet
 * Actual Value: odometry.safety_brake
 * Expected Value: buf[i + idPosition + 1]
 *
 */
void test_handle_odometry_packet_safety_brake(void) {

    TEST_ASSERT_EQUAL_UINT8(testData.uint8_value[1], testData.uint8_value[0]);
}
```

Figure 15 Function with Arguments Test 1

In other cases, it is an array of values incremented or pushed through some loop, for these cases, the values are added instead of dynamically assigning them and test their sums as an assertion, since I am not aware of the incoming data length and nature which is in most cases, live readings from some CAN or Ethernet connection.

```
void handle_odometry_packet(char *buf) {
    uint8_t idType;                     // ID value
    uint8_t idPosition;             // Byte location from where to look for ID
    volatile uint8_t j = 3;

    // Find velocity
    idType = MSG_TYPE_VELOCITY;
    idPosition = 0; // Which byte is velocity ID
    if (memcmp(&buf[idPosition], &idType, 1) == 0) {
        for(int i = 0; i<sizeof(float); i++) {          // Extract velocity data
            odometry.velocity[j--] = buf[i+idPosition+1];

            /**
             * Testing Area -- collect totals instead if looping
             */
            test_data_init();

            testData.string_value[0] += buf[i + idPosition + 1];
            testData.string_value[1] += odometry.velocity[j--];
        }
    }
```

Figure 16 Function with Arguments 2 [18]

```
/**
 * Test: handle_odometry_packet_st_wheel_angle
 * Target method: handle_odometry_packet
 * Parent method: udp_serve
 *
 * @brief Testing the sum of assigned array values in odometry struct after calling handle_odometry_packet
 * Actual Value: odometry.st_wheel_angle
 * Expected Value: buf[i + idPosition + 1]
 *
 */
void test_handle_odometry_packet_velocity(void) {

    TEST_ASSERT_EQUAL_STRING(testData.string_value[3],
            testData.string_value[2]);
}
```

Figure 17 Function with Arguments test 2

4.3 Interrupt Service Routine Functions Tests

Interrupt Service Routine (ISR) functions proved to be tricky to capture at the needed time to get correct readings, since they are always called by the RTOS scheduler and once the scheduler started, all other execution ceases and I cannot invoke testing unless through a scheduler task. ISR constrained environment also prevents me from adding too much testing related functionality inside, since everything inside this routine should be lightweight, and quickly executed. Therefore, I create a custom RTOS task for testing purposes and call it using semaphores from within the ISR that I need to test, and for passing the data values needed to be tested I use the globally declared testData structure.

In Figure 18 an ISR callback function that is invoked by RTOS scheduler whenever a CAN data transmission is completed and inside the CAN data is analyzed and actions

taken based on these analysis, almost all the actions are simple thread calls to start separate threads outside the ISR responsible for more complex executions. Figure 19 shows the testing thread task runner which is in the main runner.c file and its main responsibility is calling the ISR test suites in the correct time whenever the task is called from ISR, these tests are then executed in their respective test suites and the results are printed outside the ISR adding no extra processing load on such states as we can see in Figure 20

```
void HAL_CAN_RxCpltCallback(CAN_HandleTypeDef* CanHandle) {
    long            task_woken = 0;
    uint8_t     data_length = 0;
    uint32_t    data_id = 0;
    uint8_t     data_id_type = 0;


    data_length = CanHandle->pRxMsg->DLC;
    data_id         = CanHandle->pRxMsg->StdId;
    data_id_type= CanHandle->pRxMsg->IDE;
// Forward message CAN1 -> ETH
if ((data_id == CAN_ID_HANDBRAKE)||(data_id == CAN_ID_WHEEL_POSITION)) {
    xSemaphoreGiveFromISR(CANBinarySemHandle, &task_woken);     // Execute CAN thread
    if (task_woken) {
        portYIELD_FROM_ISR(task_woken);
    }
    HAL_GPIO_TogglePin(LD3_GPIO_Port, LD3_Pin);
}


} else if (CanHandle->Instance == CAN2) {
    // Save CAN2 received data into structure
    Store_Miev_Data(&hcan2, data_length, data_id);

    /*
     * Testing Area (taking I/O values and testing them)
     *
     * */
    test_data_init();
    testData.uint32_value[0] = data_id;
    testData.uint32_value[1] = CanHandle->pRxMsg->StdId;
    testData.uint8_value[0] = data_length;
    testData.uint8_value[1] = CanHandle->pRxMsg->DLC;
    testData.uint8_value[2] = data_id_type;
    testData.uint8_value[3] = CanHandle->pRxMsg->IDE;

    xSemaphoreGiveFromISR(test_sem_handle_can, &task_woken);// Execute Testing thread
    if (task_woken) {
        portYIELD_FROM_ISR(task_woken);
    }
```

Figure 18 ISR Function example [18]

```
void test_task_runner(void *p) {

    while (1) {
        if (xSemaphoreTake(test_sem_handle_can, portMAX_DELAY)) {
            printf(
                    "\n-----------------------------------UNITY ISR TESTING START--------------------
            UNITY_BEGIN();
            RUN_TEST(test_HAL_CAN_RxCpltCallback_data_id);
            RUN_TEST(test_HAL_CAN_RxCpltCallback_data_id_type);
            RUN_TEST(test_HAL_CAN_RxCpltCallback_data_length);

            UNITY_END();

            printf(
                    "-------------------------UNITY ISR TESTING END-------------------------\n\n");
        }
    }
}
```

Figure 19 ISR Function Task

```
/**
 * Test: HAL_CAN_RxCpltCallback_data_id
 * Target method: HAL_CAN_RxCpltCallback
 * Parent method:
 *
 * @brief Testing the assigned data values in call back function of receiving CAN messages
 * Actual Value: data_id which is stored in testData.uint32_value[0]
 * Expected Value: CanHandle->pRxMsg->StdId which is stored in testData.uint32_value[1]
 *
 */
void test_HAL_CAN_RxCpltCallback_data_id(void) {
    TEST_ASSERT_EQUAL_UINT32(testData.uint32_value[1],
            testData.uint32_value[0]);
}
 * Test: HAL_CAN_RxCpltCallback_data_id_type
void test_HAL_CAN_RxCpltCallback_data_id_type(void) {
    TEST_ASSERT_EQUAL_UINT8(testData.uint8_value[3], testData.uint8_value[2]);
}
 * Test: HAL_CAN_RxCpltCallback_data_id_length
void test_HAL_CAN_RxCpltCallback_data_length(void) {
    TEST_ASSERT_EQUAL_UINT8(testData.uint8_value[1], testData.uint8_value[0]);
}
```

Figure 20 ISR Function Test

With this approach I am able to invoke the needed tests at the correct time frame where the values being manipulated still resides in their respective memory addresses, these values are copied to my structure and their validity is tested elsewhere.

4.4 Inapplicable Functions

There are functions that cannot be tested for various reasons. Some functions are not yet completed, some are not fully implemented, when tested they resulted in strange behavior and sometimes system crashes. Other functions are deemed not important and I am advised to ignore them.

```
// TBD
uint8_t CAN2_Rx(void) {
    while(!CAN2->RF0R&CAN_RF0R_FMP0);   // Wait for a message complete in FIFO0
    uint8_t RxD = (CAN2->sFIFOMailBox[0].RDLR) & 0xFF;  // Read data bytes
    CAN2->RF0R |= CAN_RF0R_RFOM0;       // Release FIFO0 output mailbox
    return RxD;
}
```

Figure 21 Inapplicable Function 1 [18]

In some cases, the tests fail to execute due to the fact that this function gets its input from CAN data stream and since I have no live data stream, I will skip these kinds of functions. As a future solution, two boards with CAN and Ethernet connections should be used to mimic the actual environment the software is supposed to work in.

```
void test_CAN2_Rx(void) {
    uint8_t retval = CAN2_Rx();
    TEST_ASSERT_EQUAL_UINT8_MESSAGE(retval,
            ((CAN2->sFIFOMailBox[0].RDLR) & 0xFF), "Test_CAN2_Rx_1 Failed");
}
```

Figure 22 Inapplicable Function 1 Test

Other functions are not tested because their purpose is to pass data to some other function and there are no return values from these function calls, so no way of testing what happens there (to my knowledge).

```
void ethernetif_notify_conn_changed(struct netif *netif) {
    MX_LWIP_Init();
    User_notification(&gnetif);
}
```

Figure 23 Inapplicable Function 2 [18]

Finally, other functionality is not tested for lack of time during this project, but same approach can be followed and the system can be fully tested in the future. It took the larger span of the thesis time to check the frameworks, find the one to use and set it up.

5 Results and Analysis

Using the TeraTerm© [16] application it is possible to get the printing output on its terminal and also using the logging feature that comes with it, I am able to log the results of each test run into a text file and save it for reference. In total during the time this thesis is written there are 32 functions, 14 were tested, and 18 skipped due to the lack of time, the means to test them, as per advice from the implementation team or they are not yet completed.

Module	**Functions**	**Covered**	**Skipped**	**Coverage**
app_ethernet	2	1	1	
can	8	6	2	
ethernetif	2	2	0	
main	2	2	0	
ros	2	2	0	
rtc	4	1	3	
server_netconn	5	0	5	
udpclient	5	0	5	
utils	2	0	2	
	32	14	18	43.75%

Table 3 Testing Coverage

As a mean of synchronization between the implementation team and testing, a requirements/testing register was created and is being updated by implementation whenever a new functionality is added to the requirements, and for each one a test suite is created or to be created in the future.

Group	Requirements	Functionality	Devel. status	Responsible	Test state	Test result	Tested by	Test date	Comments/ Reccomendations
ROS - iMiev	Recieve all iMiev messages over CAN2 bus		Finished		Finished	3 Passed	Eslam	25.04.2018	Called from ISR callback function by inv
ROS - iMiev	Store all messages from iMiev to corresponding data structure		Finished		Finished	8 Pass 2 Fail	Eslam	24.04.2018	Stored I/O in data struct and tested then
Drive - ROS	Recieve all drive controller messages over CAN1 bus		Finished		Finished	3 Passed	Eslam	25.04.2018	Stored I/O in data struct and tested then
Drive - ROS	Store all messages froim drive controller to corresponding data structure		Under development						
Drive - ROS	Recieve UDP packet from ROS containing odometry information		Finished		Pending				need to know why function does that
Drive - ROS	Store odemetry information from ROS to corresponding data structure		Finished		Finished	3 Passed	Eslam		Stored I/O in data struct and tested then
Drive - ROS	Construct CAN message based on odometry information recieved from ROS		Finished		Pending				need to know why function does that
Drive - ROS	Forward odometry CAN messages to drive controller over CAN1 bus		Finished						
ROS - iMiev	Each CAN message that is sent out from controller has last byte counter that is increased after every sending and re-initialized after overflow. If massge length is already 8 bytes then counter will not be added.		Not implemented						
iMiev - Drive	Forward velocity, steering angle and gear information from iMiev CAN to drive controller	Construct CAN messages with steering angle, velocity and gear infromation and forward them to drive controller over CAN1 bus	Finished						
ROS - iMiev	Send out UDP packet to ROS periodically every x ms and after recieving certain CAN messages from drive controller	Construct a UDP packet from iMiev and drive controller messages. Forward UDP packet to ROS over Ethernet	Under development						

Figure 24 Requirements/Tests register

5.1 Tests Results

The test results are consolidated into report files; each function is broken down into segments each one covering one variable that is being tested.

5.1.1 **Main Module**

All tests passed

```
-----START || TESTING: main.c ----------
../Tests/Src/runner.c:90:test_MPU_Config_RNR:PASS
../Tests/Src/runner.c:91:test_MPU_Config_RBAR:PASS
../Tests/Src/runner.c:92:test_MPU_Config_RASR:PASS
../Tests/Src/runner.c:93:test_CPU_CACHE_Enable_SCB_EnableICache_ICIALLU:PASS
../Tests/Src/runner.c:94:test_CPU_CACHE_Enable_SCB_EnableICache_CCR:PASS
../Tests/Src/runner.c:95:test_CPU_CACHE_Enable_SCB_EnableDCache_CSSELR:PASS
../Tests/Src/runner.c:96:test_CPU_CACHE_Enable_SCB_EnableDCache_CCR:PASS
-----END || TESTING: main.c ----------
```

Figure 25 Test Report: Main Module

5.1.2 **App_Ethernet Module**

2 tests passed and 2 failed

```
-----START || TESTING: app_ethernet.c ----------
../Tests/Src/runner.c:33:test_User_notification_LD1_SET:FAIL: Expected 1 Was 0
../Tests/Src/runner.c:101:test_User_notification_LD1_RESET:PASS
../Tests/Src/runner.c:65:test_User_notification_LD3_SET:FAIL: Expected 16384 Was 0
../Tests/Src/runner.c:103:test_User_notification_LD3_RESET:PASS
-----END || TESTING: app_ethernet.c ----------
```

Figure 26 Test Report: App_Ethernet Module

5.1.3 **CAN Module**

26 tests passed, 2 failed

```
-----START || TESTING: can.c ----------
../Tests/Src/runner.c:107:test_CAN1_Config_Filters_StdId:PASS
../Tests/Src/runner.c:108:test_CAN1_Config_Filters_ExtId:PASS
../Tests/Src/runner.c:109:test_CAN1_Config_Filters_RTR:PASS
../Tests/Src/runner.c:110:test_CAN1_Config_Filters_IDE:PASS
../Tests/Src/runner.c:111:test_CAN1_Config_Filters_DLC:PASS
../Tests/Src/runner.c:112:test_CAN2_Config_Filters_StdId:PASS
../Tests/Src/runner.c:113:test_CAN2_Config_Filters_ExtId:PASS
../Tests/Src/runner.c:114:test_CAN2_Config_Filters_RTR:PASS
../Tests/Src/runner.c:115:test_CAN2_Config_Filters_IDE:PASS
../Tests/Src/runner.c:116:test_CAN2_Config_Filters_DLC:PASS
../Tests/Src/runner.c:117:test_CAN3_Config_Filters_StdId:PASS
../Tests/Src/runner.c:118:test_CAN3_Config_Filters_ExtId:PASS
../Tests/Src/runner.c:119:test_CAN3_Config_Filters_RTR:PASS
../Tests/Src/runner.c:120:test_CAN3_Config_Filters_IDE:PASS
../Tests/Src/runner.c:121:test_CAN3_Config_Filters_DLC:PASS
../Tests/Src/runner.c:248:test_Store_Miev_Data_iMievData_update_status:FAIL: Expected 1 Was 0
../Tests/Src/runner.c:262:test_Store_Miev_Data_iMievTime_update_status:FAIL: Expected 1 Was 0
../Tests/Src/runner.c:124:test_Store_Miev_Data_iMievData_ready:PASS
../Tests/Src/runner.c:125:test_Store_Miev_Data_iMievTime_ready:PASS
../Tests/Src/runner.c:126:test_Store_Miev_Data_iMievData_key:PASS
../Tests/Src/runner.c:127:test_Store_Miev_Data_iMievTime_key:PASS
../Tests/Src/runner.c:128:test_Store_Miev_Data_iMievData_switch_cases:PASS
../Tests/Src/runner.c:129:test_Store_Miev_Data_iMievTime_switch_cases:PASS
../Tests/Src/runner.c:130:test_Store_Miev_Data_iMievData_update_status_2:PASS
../Tests/Src/runner.c:131:test_Store_Miev_Data_iMievTime_update_status_2:PASS
../Tests/Src/runner.c:132:test_CAN_TX_STATE:PASS
../Tests/Src/runner.c:133:test_CAN_TX_type:PASS
../Tests/Src/runner.c:134:test_CAN_TX_size:PASS
-----END || TESTING: can.c ----------
```

Figure 27 Test Report: CAN Module

5.1.4 **ROS Module**

All tests passed

```
-----START || TESTING: ros.c ----------
../Tests/Src/runner.c:141:test_Construct_ROS_Packet_can_id:PASS
../Tests/Src/runner.c:142:test_Construct_ROS_Pack_ready_time:PASS
../Tests/Src/runner.c:143:test_Construct_ROS_Packet_ready_data:PASS
../Tests/Src/runner.c:144:test_Construct_ROS_Packet_key_can_id:PASS
../Tests/Src/runner.c:145:test_Construct_ROS_Packet_key_time:PASS
../Tests/Src/runner.c:146:test_Construct_ROS_Packet_key_data:PASS
../Tests/Src/runner.c:147:test_Construct_ROS_Packet_charger_temp_can_id:PASS
../Tests/Src/runner.c:148:test_Construct_ROS_Packet_charger_temp_time:PASS
../Tests/Src/runner.c:149:test_Construct_ROS_Packet_charger_temp_data:PASS
../Tests/Src/runner.c:150:test_Construct_ROS_Packet_motor_temp_rpm_can_id:PASS
../Tests/Src/runner.c:151:test_Construct_ROS_Packet_motor_temp_rpm_time:PASS
../Tests/Src/runner.c:152:test_Construct_ROS_Packet_motor_temp_rpm_data:PASS
../Tests/Src/runner.c:153:test_Construct_ROS_Packet_driving_range_can_id:PASS
../Tests/Src/runner.c:154:test_Construct_ROS_Packet_driving_range_time:PASS
../Tests/Src/runner.c:155:test_Construct_ROS_Packet_driving_range_data:PASS
../Tests/Src/runner.c:156:test_Construct_ROS_Packet_amps_volts_can_id:PASS
../Tests/Src/runner.c:157:test_Construct_ROS_Packet_amps_volts_time:PASS
../Tests/Src/runner.c:158:test_Construct_ROS_Packet_amps_volts_data:PASS
../Tests/Src/runner.c:159:test_Construct_ROS_Packet_charging_status_can_id:PASS
../Tests/Src/runner.c:160:test_Construct_ROS_Packet_charging_status_time:PASS
../Tests/Src/runner.c:161:test_Construct_ROS_Packet_charging_status_data:PASS
../Tests/Src/runner.c:162:test_Construct_ROS_Packet_ac_amps_volts_can_id:PASS
../Tests/Src/runner.c:163:test_Construct_ROS_Packet_ac_amps_volts_time:PASS
../Tests/Src/runner.c:164:test_Construct_ROS_Packet_ac_amps_volts_data:PASS
../Tests/Src/runner.c:165:test_Construct_ROS_Packet_ac_status_can_id:PASS
../Tests/Src/runner.c:166:test_Construct_ROS_Packet_ac_status_time:PASS
../Tests/Src/runner.c:167:test_Construct_ROS_Packet_ac_status_data:PASS
../Tests/Src/runner.c:168:test_Construct_ROS_Packet_gas_can_id:PASS
../Tests/Src/runner.c:169:test_Construct_ROS_Packet_gas_time:PASS
../Tests/Src/runner.c:170:test_Construct_ROS_Packet_gas_data:PASS
../Tests/Src/runner.c:171:test_Construct_ROS_Packet_break_pedal_can_id:PASS
```

Figure 28 Test Report: ROS Module

5.1.5 **Ethernetif Module**

2 tests passed, 1 ignored

```
-----START || TESTING: ethernetif.c ----------
../Tests/Src/runner.c:192:test_sys_jiffies:PASS
../Tests/Src/runner.c:193:test_sys_now:PASS
../Tests/Src/runner.c:50:test_ethernetif_notify_conn_changed:IGNORE: Method not implemented
-----END || TESTING: ethernetif.c ----------
```

Figure 29 Test Report: Ethernetif Module

5.1.6 **RTC Module**

1 test passed

```
-----START || TESTING: rtc.c ----------
../Tests/Src/runner.c:198:test_RTC_CalendarConfig:PASS
-----END || TESTING: rtc.c ----------
```

Figure 30 Test Report: RTC Module

5.1.7 **Normal Tests Summary**

Note that Unity consider the end result as a failure if one or more tests failed

```
-----------------------
91 Tests 4 Failures 1 Ignored
FAIL

-----------------------------------UNITY TESTING END-----------------------------------
```

Figure 31 Test Report: Normal Tests Summary

5.1.8 **Interrupt Service Routine Tests**

These tests are consolidated in a different task function that is called later when the scheduler calls the ISR function that is to be tested

3 tests passed

```
-----------------------------------UNITY ISR TESTING START----------------
../Tests/Src/runner.c:52:test_HAL_CAN_RxCpltCallback_data_id:PASS
../Tests/Src/runner.c:53:test_HAL_CAN_RxCpltCallback_data_id_type:PASS
../Tests/Src/runner.c:54:test_HAL_CAN_RxCpltCallback_data_length:PASS

----------------------
3 Tests 0 Failures 0 Ignored
OK
-------------------------UNITY ISR TESTING END-------------------------
```

Figure 32 Test Report: ISR Functions Tests

5.2 Detected bugs and Possible Code Improvements

In total there are 4 failed tests from 94 test cases, surely more problems would be uncovered with more tests and with live readings from the car that is being operated by the software in development.

5.2.1 **Failed Tests**

File: app_ethernet

The function (User_notification) objective as per the documentation, is to notify the user about the network interface configuration status.

```
if (netif_is_up(netif)) {
  /* Turn On LED 1 to indicate ETH and LwIP init success*/
  HAL_GPIO_WritePin(LD1_GPIO_Port, LD1_Pin, GPIO_PIN_SET);
      HAL_GPIO_WritePin(LD3_GPIO_Port, LD3_Pin, GPIO_PIN_RESET);
  } else {
 /* Turn On LED 3 to indicate ETH and LwIP init error */
      HAL_GPIO_WritePin(LD3_GPIO_Port, LD3_Pin, GPIO_PIN_SET);
      HAL_GPIO_WritePin(LD1_GPIO_Port, LD1_Pin, GPIO_PIN_RESET);
  }
```

Figure 33 User Notification Function [18]

In order to test this functionality, each one of the four HAL_GPIO_WritePin function calls are tested separately, viewing the actual login behind this function it shows that it assign a value to a certain LD Pin

```c
void HAL_GPIO_WritePin(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin,
{
  /* Check the parameters */
  assert_param(IS_GPIO_PIN(GPIO_Pin));
  assert_param(IS_GPIO_PIN_ACTION(PinState));

  if(PinState != GPIO_PIN_RESET)
  {
    GPIOx->BSRR = GPIO_Pin;
  }
  else
  {
    GPIOx->BSRR = (uint32_t)GPIO_Pin << 16;
  }
```

Figure 34 Function called by User Notification Function [18]

So I call the function in each test case and then check the new value assigned in the GPIO_Pin variable

```c
/**
 * Test: User_notification_LD1_SET
 * Target method: User_notification
 * Parent method: ethernetif_notify_conn_changed
 *
 * @brief Testing the assigned values in LD1_GPIO_Port struct after calli
 *        method with GPIO_PIN_SET
 * Actual Value: LD1_GPIO_Port->BSRR
 * Expected Value: LD1_Pin
 *
 */
void test_User_notification_LD1_SET(void) {
    HAL_GPIO_WritePin(LD1_GPIO_Port, LD1_Pin, GPIO_PIN_SET);
    TEST_ASSERT_EQUAL_UINT16(LD1_Pin, LD1_GPIO_Port->BSRR);
}
```

Figure 35 Failed Test of LD1 Variable with GPIO_PIN_SET variation

However, the returned value is not equal to the value assigned in the LD1_GPIO_Port->BSRR.

The second failed case in app_ethernet is of the same function but a different LD pin

```c
/**
 * Test: User_notification_LD3_RESET
 * Target method: User_notification
 * Parent method: ethernetif_notify_conn_changed
 *
 * @brief Testing the assigned values in LD1_GPIO_Port struct after calling HAL_GPIO_WritePin
 *        method with GPIO_PIN_RESET
 * Actual Value: LD1_GPIO_Port->BSRR
 * Expected Value: ((uint32_t)LD3_Pin << 16)
 *
 */
void test_User_notification_LD3_RESET(void) {
    HAL_GPIO_WritePin(LD3_GPIO_Port, LD3_Pin, GPIO_PIN_RESET);
    TEST_ASSERT_EQUAL_UINT16(((uint32_t)LD3_Pin << 16), LD3_GPIO_Port->BSRR);
}
```

Figure 36 Failed Test of LD3 with GPIO_PIN_RESET

Same as the first failed case, although the same logic applied to the first ASSERT parameter as the original HAL_GPIO_WritePin function, still the values are not the same.

File: can

There are 2 failed cases in the can module, both are in the Store_Miev_Data function. Which has the objective of saving incoming CAN data into the structure.

The first case is testing the structure iMievData variable update_status, that it contains the correct value of STAT_UPDATING.

```
iMievData.update_status = STAT_UPDATING;
iMievTime.update_status = STAT_UPDATING;

testData.uint8_value[0] = iMievData.update_status;
testData.uint8_value[1] = iMievTime.update_status;
```

Figure 37 Store_Miev_Data Function(part) [18]

The values are added to the testData structure created by me for moving the test data that are not globally accessible outside the module.

```
/**
 * Test: Store_Miev_Data_iMievData_update_status
 * Target method: Store_Miev_Data
 * Parent method:
 *
 * @brief Testing the assigned values in iMievData struct after calling Store_Miev_Data
 * Actual Value: iMievData.update_status which is stored in testData.uint8_value[0]
 * Expected Value: STAT_UPDATING
 *
 */
void test_Store_Miev_Data_iMievData_update_status(void) {
    TEST_ASSERT_EQUAL_UINT8(STAT_UPDATING, testData.uint8_value[0]);
}
```

Figure 38 Failed Test of iMievData.update_status and STATE_UPDATING

The values do not match when tested, it is probably because there is no actual data stream incoming to be saved while testing.

The second case was similar to the first, testing that STAT_UPDATING value is saved in the iMievTime structure variable update_status.

```
/**
 * Test: Store_Miev_Data_iMievTime_update_status
 * Target method: Store_Miev_Data
 * Parent method:
 *
 * @brief Testing the assigned values in iMievTime struct after calling Store_Miev_Data
 * Actual Value: iMievTime.update_status which is stored in testData.uint8_value[1]
 * Expected Value: STAT_UPDATING
 *
 */
void test_Store_Miev_Data_iMievTime_update_status(void) {
    TEST_ASSERT_EQUAL_UINT8(STAT_UPDATING, testData.uint8_value[1]);
}
```

Figure 39 Failed Test of iMievTime.update_status and STATE_UPDATE

Same as case 3, the value assigned does not match the expected and it is also probably due to the lack of the actual data stream that is supposed to be incoming in the live environment.

It is noted that some modules are entirely created by the implementation teams, for example the modules **udpclient, ros,** and **utils.** if there is a possibility to replace some of these modules with auto generated code using STMCubeMx or something similar, it would serve to reduce the possibility of human errors, thus reducing the overall percentage of software bugs.

5.3 Inapplicable Tests Alternatives

For the functions that are not tested, it is advised that access to the live environment would be granted to the future testing teams in order to have access to live readings from the car's sensors and outputs. With this kind of information, it would be possible to test all the functions that depends on incoming data from CAN or Ethernet connections, validate readings coming from the car and going into the car's control points and test more thoroughly different cases and possibilities that the car might face in normal/abnormal driving conditions.

Regarding the functions that contained input arguments and are tested using global data structure to copy data to the test environment, a less intrusive approach would be to create mock data examples of all the arguments as well as mocks for the expected outputs of these functions. That would require the implementation teams to provide these examples during the implementation itself and document them for later use by the testing teams.

6 Summary

The main purpose of this thesis was to establish a unit testing environment for the software being developed, test as much as possible of the existing code, and provide a portable way of adding further test suites for newly developed modules without adding too much intrusion on the software.

The framework chosen is open source, flexible and can be redesigned for more complex purposes in the future in case the current features proves to be insufficient to cover all possible test cases.

The test module has been created as a separate entity that can be added to any other version of the software with minimal changes required, the test suites calls are consolidated in one main runner file for easier enabling/disabling of specific tests.

The tests have been conducted in a non-live environment using a development board same as the ones being used in the actual project, the results obtained from these tests shows that the code is working properly in most cases, the data is being transferred as intended and there are minor cases of failures that are clearly stated and logged by the framework.

The installed test module had negligible effect on the execution performance of the main project, and that it will have similar effect during live tests as well.

Acknowledgments

This work was supported by the Department of Computer Systems of the Tallinn University of Technology.

References

[1] S. Labs, “Quality Assurance and Software Testing: A Brief History,” 12 July 2016. [Online]. Available: https://saucelabs.com/blog/quality-assurance-and-software-testing-a-brief-history. [Accessed 20 April 2018].

[2] S. P. A. R. M. a. G. C. J. Karmore, “Development of Software Interface for Testing of Embedded System,” *2013 15th International Conference on Advanced Computing Technologies (ICACT),*, pp. 1-6, 2013.

[3] A. Berger, “The Basics of Embedded Software Testing: Part 1,” 07 February 2011. [Online]. Available: https://www.embedded.com/design/other/4212929/2/The-basics-of-embedded-software-testing--Part-1. [Accessed 10 February 2018].

[4] “Project's Google Drive,” 29 April 2018. [Online]. Available: https://drive.google.com/drive/folders/0B9SJu3UyveivdmVNTFB2SGlMMTA.

[5] “Types of Non Functional Software Testing,” 1 February 2012. [Online]. Available: https://www.testing-whiz.com/blog/types-of-non-functional-software-tests. [Accessed 17 February 2018].

[6] A. Berger, “The Basics of Embedded Software Testing: Part 2,” 07 February 2011. [Online]. Available: https://www.embedded.com/design/other/4212937/The-basics-of-embedded-software-testing--Part-2-. [Accessed 16 February 2018].

[7] J. Ganssle, Embedded Systems: World Class Designs, Oxford: Newnes, 2007.

[8] “Tunghai University - DOE_Project,” [Online]. Available: http://www2.thu.edu.tw/~emtools/DOE_project/NTCU/ppt_to_%AAF%AE%FC/Overview.ppt. [Accessed 15 March 2018].

[9] “The FreeRTOS™ Kernel. Market Leading, De-facto Standard and Cross Platform RTOS kernel,” 25 April 2018. [Online]. Available: https://www.freertos.org/.

[10] “Home Page,” [Online]. Available: http://www.openstm32.org/HomePage. [Accessed 20 March 2018].

[11] “Autoconf,” 28 April 2018. [Online]. Available: http://www.gnu.org/software/autoconf/autoconf.html.

[12] “Home Page,” 15 March 2018. [Online]. Available: http://www.throwtheswitch.org/.

[13] “ThrowTheSwitch/Unity Simple Unit Testing for C,” ThrowTheSwitch, [Online]. Available: https://github.com/ThrowTheSwitch/Unity. [Accessed 8 January 2018].

[14] “Home - STMicroelectronics,” STMicroelectronics, [Online]. Available: http://www.st.com/content/st_com/en.html. [Accessed 24 April 2018].

[15] “Nucleo STM32F767,” 28 April 2018. [Online]. Available: https://www.ittgroup.ee/1602-large_default/nucleo-stm32f767.jpg.

[16] “Tera Term Home Page,” 25 April 2018. [Online]. Available: https://ttssh2.osdn.jp/index.html.en.

[17] “UM1974 User manual STM32 Nucleo-144 boards,” 28 December 2017. [Online]. Available: http://www.st.com/content/ccc/resource/technical/document/user_manual/group0/26/49/90/2e/33/0d/4a/da/DM00244518/files/DM00244518.pdf/jcr:content/translations/en.DM00244518.pdf.

[18] “TTÜ iseauto projekti master kontrolleri embedded soft,” 27 April 2018. [Online]. Available: https://gitlab.pld.ttu.ee/iseauto/master_controller.