TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies
Department of Computer Systems

Kristjan Harri Laur 178193IASM

EMULATING AN UGV ON A MODEL TANK

Master Thesis

Academic Supervisor
Peeter Ellervee
PhD
Supervisor
Artti Zirk

Tallinn 2020

TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond
Arvutisüsteemide instituut

Kristjan Harri Laur 178193IASM

UGV EMULEERIMINE MUDELTANKIL

Magistritöö

Akadeemiline juhendaja
Peeter Ellervee
PhD
Juhendaja
Artti Zirk

Tallinn 2020

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: Kristjan Harri Laur .....
(signature)

Date: August 05, 2020

Annotatsioon

Antud lõputöö eesmärgiks on luua töötav mudel tank, mis emuleeriks täismõõdus mehitamata maismaasõiduki liikumist ning baasfunktsioone. Antud töö sisaldab endas antud roboti ehitamist ning kokkupanekut, programmeerimist ning testimist.

Töö esimeses peatükis räägitakse tanki füüsilisest ehitusest ning komponentidest, mida töö käigus kasutatakse. Selle peatüki käigus selgitatakse tükkide valikust, mida valitud osad endast kujutavad ning kuidas nad üksteisega ühendatud on. Lisaks tuuakse välja ka terve süsteemi üldpilt. Lõpetatakse peatükk füüsiliste osadega seoses tekkinud probleemidega ning kuidas neid lahendati.

Lõputöö teises peatükis räägitakse roboti programmeerimisest. Teemadeks siinkohal on valitud programmeerimiskeskond ning keeled ja ükshaaval räägitakse läbi erinevate komponentide programmeerimiskäigud. Selles peatükis räägitakse süsteemi juhtimissüsteemi, sealhulgas mootorite, mitmete sisend-väljundite ning osadevahelise suhtluse kontrollist. Lisaks tuuakse välja ka programmeerimisel tekkinud vead ja parendused.

Lõputöö viimases suuremas peatükis kirjutatakse süsteemi testimisest. Testimine toimus nii komponentide tasandil, kus testiti üksikuid osasid, kui ka terve süsteemina, kui kõik ühendatud olid. Siinkohal tuuakse välja testimiste käik ning nende käigus välja tulnud vead ja kõige lõpuks räägitakse ka, mida oleks võinud paremini teha ja mis edasiarendusi võimalik teha on.

Töö lõpuks valmis kaugjuhitav mudel tank, mis suudab emuleerida päris mehitamata sõiduki liikumist ning baasfunktsioone. Kuigi antud tööle on võimalik teha mitmeid edasiarendusi ning parandusi, on antud töö käigus valmis saanud robot võimeline tegema sellele ette antud ülesandeid.

Lõputöö on kirjutatud inglise keeles ning sisaldab teksti 28. leheküljel, 4 peatükki, 13 joonist, 0 tabelit.

Abstract

The main aim of this thesis is to create a miniature version of an real life unmanned ground vehicle. The goal of this mini-vehicle is to emulate the movements and basic functions of the real thing. As a part of this, the thesis will cover the creating and assembly of the system as well as the programming and testing of it.

The first chapter will cover the general assembly as well as the physical components of the system. There is a small discussion on the choice of components and their general characteristics. The connections between the parts is also brought out and the general overview of the system is provided. The chapter ends with the problems regarding to the physical components and the fixes made.

The second chapter discusses the programming portion of the system. The topics cover the chosen development environment, languages and for each programmable component there is also a overview on what was done. The chapter covers the control algorithm's for the motors, general input/output and the communication between components. There is also a small discussion on the problems that arose and fixes made.

The last chapter covers the testing portion, which is done both on a component level as well as a whole system. The testing process and the issues that arose are discussed and how they were fixed. There is also information on what could be done better in the future and what additions could be made.

As the end result, a remote controlled model tank was made, which is able to emulate an real work unmanned vehicle's movement and basic input/output systems. Although there are plenty of improvements to be made, the robot is able to complete the tasks it is needed to do.

The thesis is in English and contains 28 pages of text, 4 chapters, 13 figures, 0 tables.

List of abbreviations and terms

UGV	Unmanned Ground Vehicle
I/O	Input/Output
USB	Universal Serial Bus
UI	User Interface
CAN	Controller Area Network
UDP	User Datagram Protocol
DCDC	Direct Current to Direct Current
I2C	Inter-Integrated Circuit
SPI	Serial Peripheral Interface
USART	Universal Synchronous and Asynchronous Receiver-Transmitter
PWM	Pulse Width Modulation
LED	Light Emitting Diode
IDE	Independent Development Environment
GPIO	General Purpose Input Output
IR	Infra-Red
ID	Identifier
HAL	Hardware Abstraction Layer

Table of Contents

List of Figures

Introduction

The following work covers the creation of an remote controlled mini-robot that mocks the movement and behaviour of an existing unmanned ground vehicle. In the work are used multiple components that will create the hardware basis for the robot. The paper will also contain a programming part, where the robots behaviour will be defined. All throughout the thesis, the author will explain the process of physical creation of the Unmanned Ground Vehicle, the programming and the testing of the system.

Background

The idea behind this work is to create a minimalist version of a UGV system. The completed work would be for use indoors and in limited space, to use for testing various parts of already existing system code and to teach new users about the UGV system through physical emulation of the movement and feedback of said system.

The current original UGV is large and can be a bit frightening to use. Not to mention, in order to learn to use the system, it is required to learn outdoors and go out to the test tracks or into other pre-defined areas. As such the weather can play quite a role in allowing or disallowing these practices to happen, as, for example, rainy weather can be quite off-putting, and as such an indoor variant would be quite welcome. The mini-tank version would also allow an easy introduction into the use of the full-scale system and provide encouragement to the future user.

Also the current introduction before using the full system is only to show the UI mock, without any systems connected and a brief introduction to the system. This mini-system would also allow for something physical to use among the introductory part.

As for the testing part, the following work would allow for easier testing, as the current procedure would is to go out and test the system out either in the garage on a stand or to go out in the field. For smaller tests, it is quite a bit of work. This system would provide enough to test out the smaller parts needed to test and to perhaps provide some insight into some of the issues that arise, all in the comfort of indoors office.

The end result would essentially be a system similar to the existing UGV, but on a much smaller scale, in which one could just drop in the same or similar code to the current one. The scope of this thesis will not go that far, however. Instead, the goal will be to make an base version of this system, upon which the rest of the mini-tank will be built upon. This will cover the main functions of movement, behaviour and connectivity, as well as some smaller input-output.

It should be noted that this system, while it will be built from ground up, is not the first version. The version built before the one in the thesis is used as a base for the tank created here, but it will be improved upon quite a bit. The differences between the systems will be also discussed in the first part of the thesis.

The goal and structure of this thesis

The main goal of this work is to create a small scale system that emulates the full-scale UGV's movement and input-output switching. To accomplish this, the system must be first built and assembled, then programmed and tested. Thus the thesis will be divided into 3 main parts:

- building and assembly of the system, which includes the selection, soldering and connecting and making the necessary changes to the system;
- programming of the system

  programming the control algorithm to operate the components on-board the system including motor and I/O control;

  programming of the CAN emulation and communications.
- testing of the system, first with a regular controller connection and later on with the operator interface, which requires changes to the programmed code, most prominent requirement being the communication over the UDP connection.

1. Creation and assembly of the hardware

This chapter covers the creation and assembly of the physical portion of the system. It will also explain the hardware differences between the first and the currently discussed version of the mini-tank.

1.1 Current solution and components analysis

As said in the introduction, there is already an existing solution, which is currently not in use. The issue with said system is that, although it is at the moment more complete, it has become a rather slowed down system thanks to its various components. This version of the robot uses an Arduino Nano microcontroller board, which uses an 8-bit and 8 MHz ATmega328 chip. Thus the processing power by now has become insufficient to support the rest of the system and is the main component that requires changing. While the existing system currently has more components that are use-able than the end result of this robot will have, then the currently built system will have more options for further development going forward than the previous version does.

To create this system, there were multiple options available to use. When choosing the components there were multiple criteria that were considered. This included physical measurements to make sure that the components could be fitted onto the system with enough room for future additions. As such, with the microcontroller board as prime example, quite a few components were rather limited in their choices.

There were, of course, other limitations as well. We needed to have a module to establish the communications with the operator and microcontroller. Thus we had to look at something that could run on a Linux system and have WIFI and USB connection capabilities, while also not using up too much power.

The microcontroller also had to have better processing power, as the the Atmega328's proved insufficient. There were consideration made for various 16-bit controllers, but in the end it was decided to make the jump straight to a 32-bit controller. The main reasoning behind this was that since the 32-bit controllers were readily available and they provide better specifications, one of them would be chosen. Choosing a 32-bit controller with

enough processign power would also help future proof the system, at least up to some extent.

With some components the situation was a lot simpler. The prime example in this case being the voltage step converters, which are required to tone down the voltage given into the other components. With these the only requirements were the possibility to turn the voltage down to either 5 or 7-12 volts and to be small enough. Our chosen converters fit exactly into these criteria and were also readily available.

1.2 The chosen components of the system

The system includes multiple different components, including the STM32 microcontroller, a Linux module, DCDC converters, a H-bridge and more. A lot of these components are either similar or the same make and model as the components on old version this work is based off of. However, there are a few differences in comparison.

Microcontroller: For the following work we use the STM32 microcontroller, which is situated on a Maple Mini board. The STM32 is a 32-bit controller using the ARM32 Cortex M3 architecture. It has 34 input/output pins, some of which can be used for PWM, analog I/O as well as for various types of communication, such as I2C, SPI and USART.[1] Although the actual chip-set allows for use of CAN protocols, the chosen Maple Mini board itself does not have a I/O port dedicated for this. The CAN pins on the STM32F103 are not directly connected to any of the pins on the Maple Mini board.To counter this the USB communication capable pins will be used with a library, which will be discussed in the programming section.

The board the controller is situated on, is, as said prior, a type of Maple Mini microcontroller board. It was one of the first microcontroller boards made and available to hobbyists and engineers. Although there are no original versions of this board made anymore, there are quite a few, mostly Chinese, companies that create either similar or copied versions of the board. The most common version of it is called a "Blue Pill." The version used in during this thesis is however the clone by a Chinese company called Baite.[2]

One of the reasons for the use of the STM32 Microcontroller on the Maple Mini is because of the need to save on space. Another reason is the use of 32-bit system as opposed to the 8-bits on the first version of the tank. The STM32 also provides a better clock speed with 72 MHz instead of the 8 MHz on the Arduino of the first version, giving a better performance and allowing for faster processing.

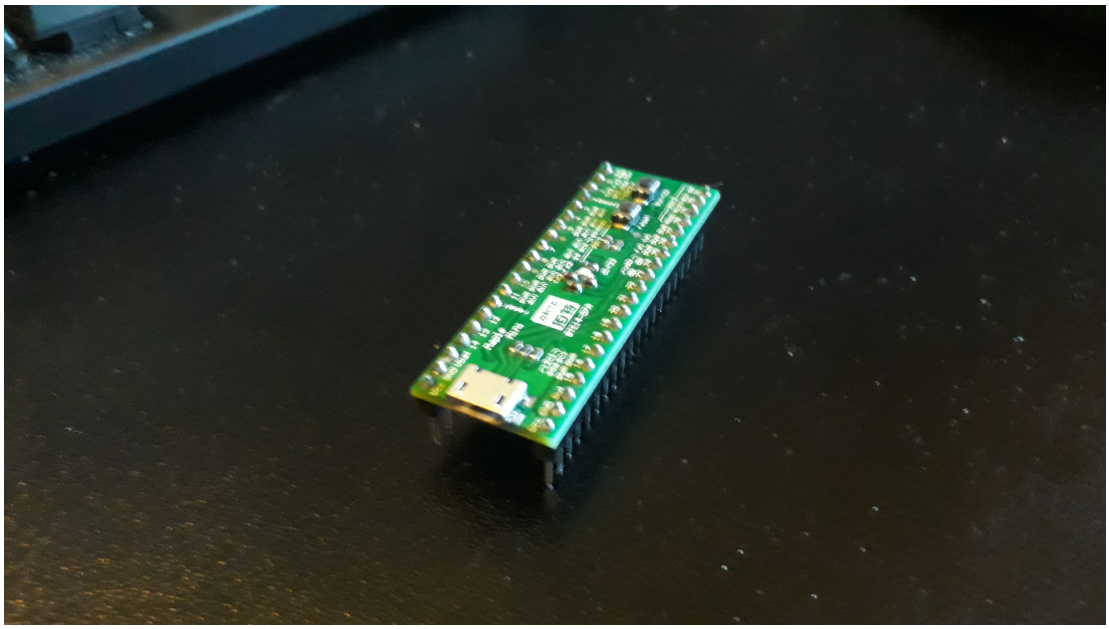

Figure 1. The Baite Maple Mini board with STM32F103C8T6 microcontroller

The main reason for the use of this controller is the aforementioned processing power. As was said, the original version of the tank used an 8MHz Arduino board, which, although was sufficient at first, became too slow. The main symptom showcasing the slowness was a serious delay in movement controls. Couple that with the addition of other inputs and outputs, it became apparent on that version that an upgrade was needed. That is the main reason why for this version the STM32 chip was chosen.

Orange Pi Zero: The Pi Zero is used as a connecting node between the operator and the STM32. It will run on an Linux Armbian distribution and be responsible for converting the incoming UDP packets from the operator into CAN messages for the microcontroller and vice-versa. Although only 4 pins total will be used on the board itself, it comes with 39 header pins, which include various input and output possibilities, including sound, IR, USB capabilities and so on.[3]

Since it will only connect to the WIFI network and run a Python code to turn the messages from UDP into CAN/USB messages and vice versa, it does not need much to work. The Pi Zero was chosen, as it provides enough and is small enough to fit onto the robot. This component is also one of the ones, that remained mainly the same, although with a couple of modifications or version differences in on-board ports.

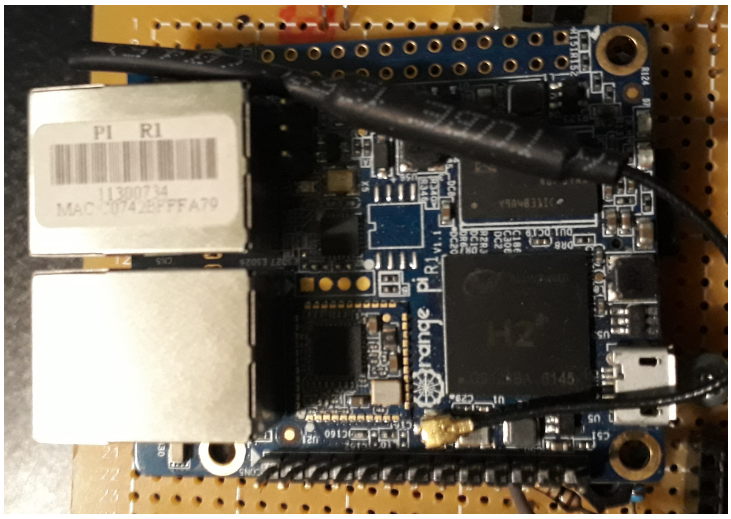

Figure 2. The Orange Pi Zero

DCDC converters: The DCDC converters or voltage down regulators will be used to change the reduce voltage levels to coming in from the batteries to use in the powering of motors through the H-Bridge and powering the microcontroller, Linux module and H-Bridges logic module. One DCDC will keep the voltage at 7-12 volts for the motors and the other will reduce the voltage down to 5V, to power the Maple Mini board, H-Bridge's logic portion and the Pi Zero.

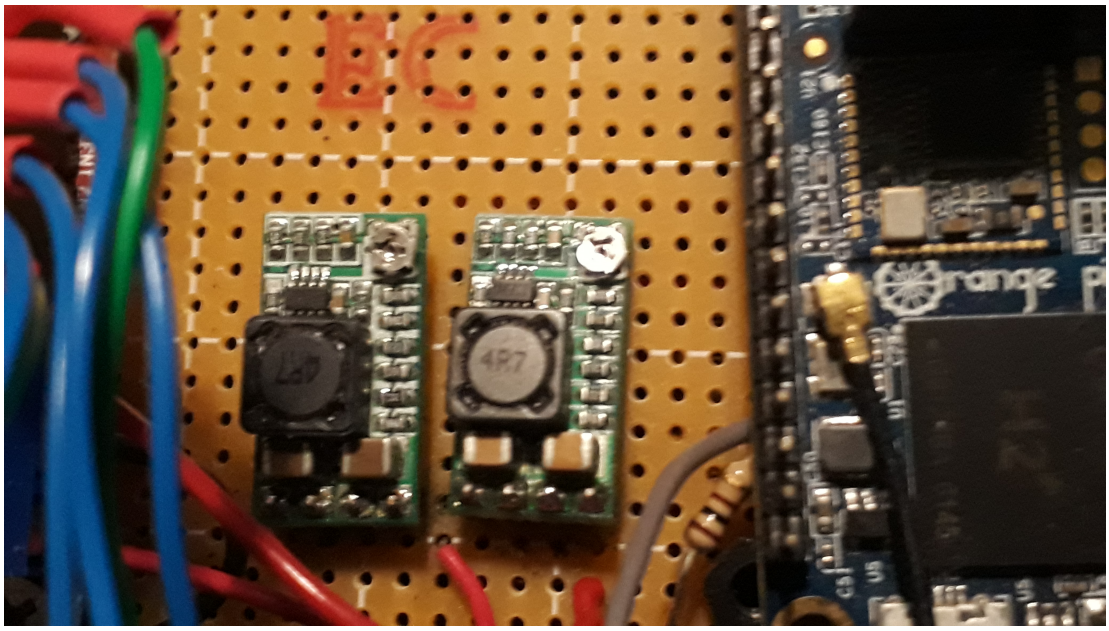

Figure 3. The DCDC Step down voltage converters

The converters themselves allow for either a fixed reduction in voltage to one of six ranges(1.8, 2.5, 3.3, 5, 9 and 12V) or adjustable range controlled by an potentiometer, with ranges from 0.8 to 17V.[4] In our case we leave one open to adjust as we see fit, while the other is modified to only allow 5 volt voltage.

H-Bridge: To provide better control over the motors, an H-bridge is used. In particular we are using the L298N Motor Driver Module, which converts the inputted PWM and direction signals into commands for the motors. It also takes the 7-12V input and, based on the inputs from the Maple Mini board, gives the motors enough power to run based on our commands. This module was chosen mainly because, at the time, it was the easiest one to acquire and thanks to its multiple directional inputs, allowed for better control over the motors.

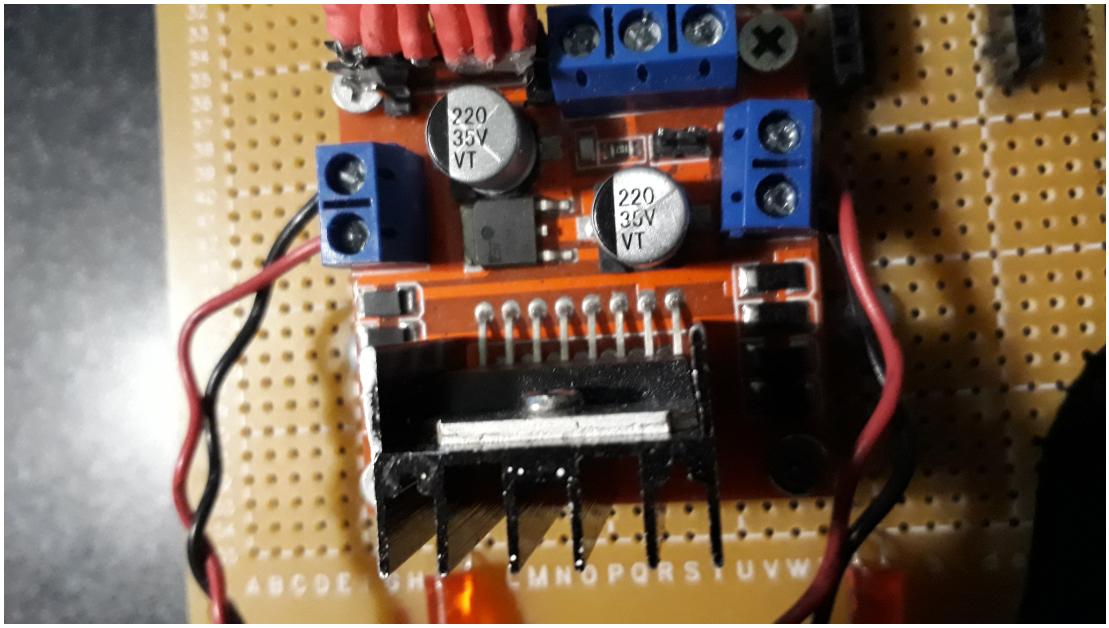

Figure 4. The L298N Motor Driver

The board itself has 6 input pins that determine the speed and direction the motors run, 2 of them being input enable pins, either ON/OFF or PWM controller, and, of the four remaining, 2 controlling one motor and 2 controlling the other. The based on the inputs to the latter pins, the direction of the motors is determined. It also has connectors to the 7-12V supply for motors, 5 V supply meant for the boards logic circuit as well as the motor connections(positive and negative) and its own ground pin.[5]

Charging board: This board is used for the to protect the batteries from overcharging and to allow the possibility of easy charging via an external 12V power source. The batteries used on the tank are 2 3.7 volt Li-Ion batteries, totalling around 7.4V for nominal voltage. The charging board itself has an logic controller on-board and thus it controls itself and says if the batteries should be charged or not or if they are done charging.

In the first version of the tank, there was no charging board and thus the batteries had to be charged separately. The board was added later, however it was placed on top of the prototyping board, thus removing some of the space on it. In our version the board itself is

placed on top of the battery box and under the prototyping board, leaving us more room to use on top and an easier way to charge the batteries without having to remove them.

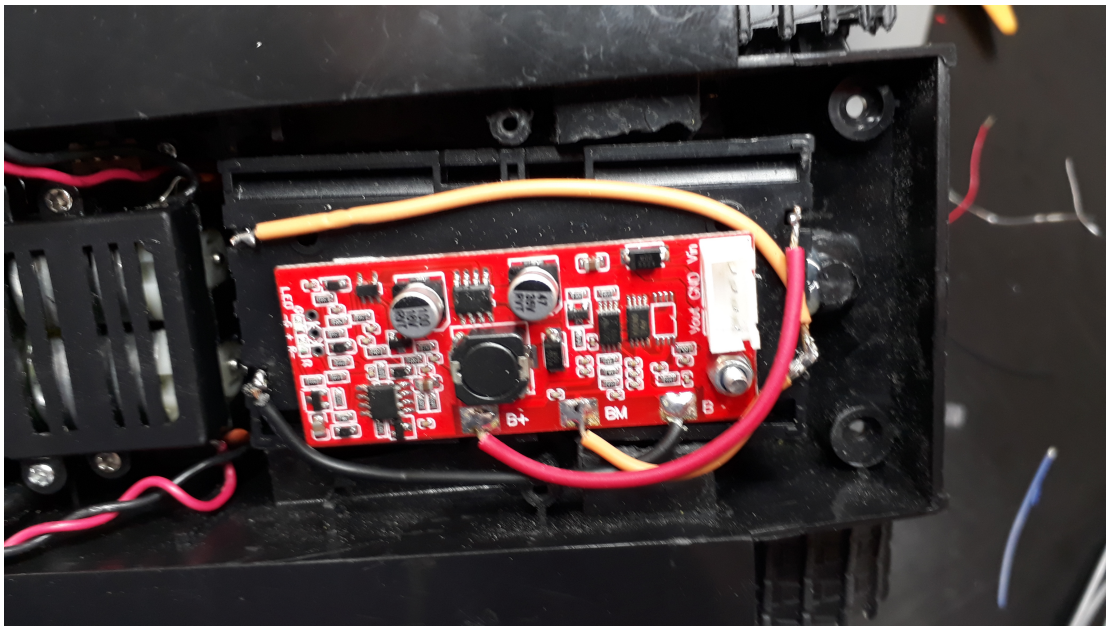

Figure 5. The intelligent charging board

Chassis and other components: The chassis and motors used for the tank are from a toy tank, based on the German WW2 Tiger tank. This was chosen as it is a easy platform to build upon. The chassis did require a few modifications, however, to accommodate a few of the other components, mainly the battery holder and the 12V charging port.

For non-electrical components we also have the prototyping board the entire system is built upon. The chosen board had to be big enough to accommodate the various components that were put on it, while also leaving enough room for future improvements and additions.

There are also some other, smaller components put on the board. For example, a simple on/off switch, to turn the whole system on and off. There are also a few LEDs installed to show the switching of the lights from the operator.

1.3 Assembly of the system

The first stage of the assembly was to figure out how to place the components onto the prototyping board. This required laying the parts onto the board to see the most optimal way to create the connections between the parts and have it be placed on the chassis. There was a requirement to save some space for future components as well, as some parts of the system would be added later on. This created an issue, where the prototyping board would not exactly fit onto the chassis. Thus some modifications was needed for the chassis.

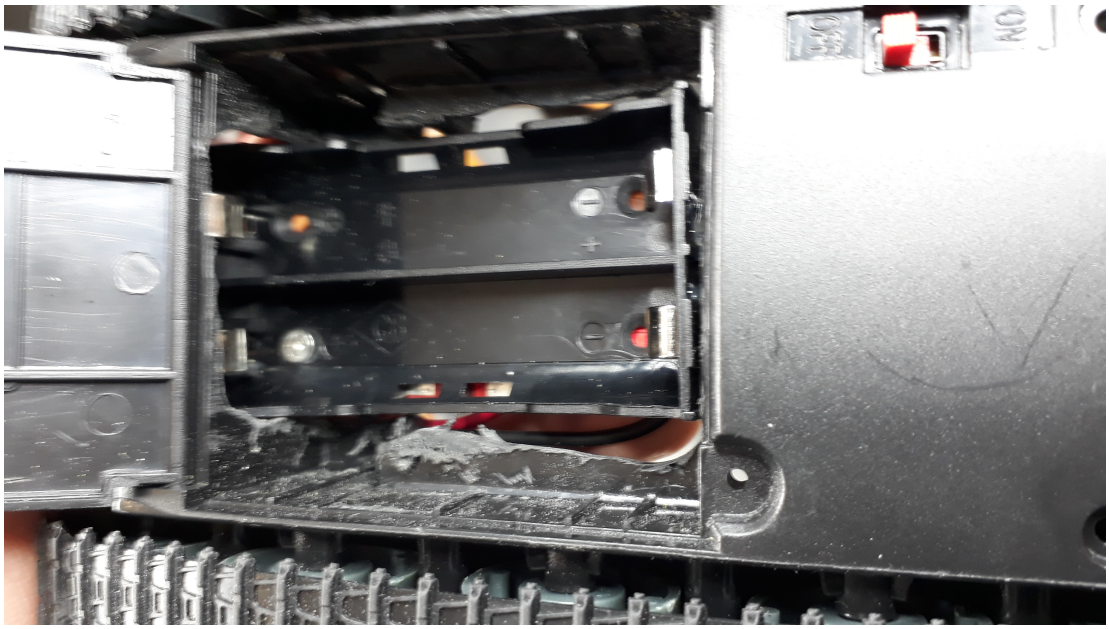

Figure 6. To fit the battery hold into the chassis, further modifications had to be made.

The bottom was cut out with a dremel to accommodate the Li-Ion battery holder and to make it easily accessible from the bottom of the tank. The battery charging protection board was also attached to the battery holder to create more room on the prototyping board itself and to have more room overall. Due to this, a small hole was also needed on the prototyping board in order to accommodate one of the Molex sockets, as otherwise the socket would be blocked by the board and access to the pins would have been problematic.

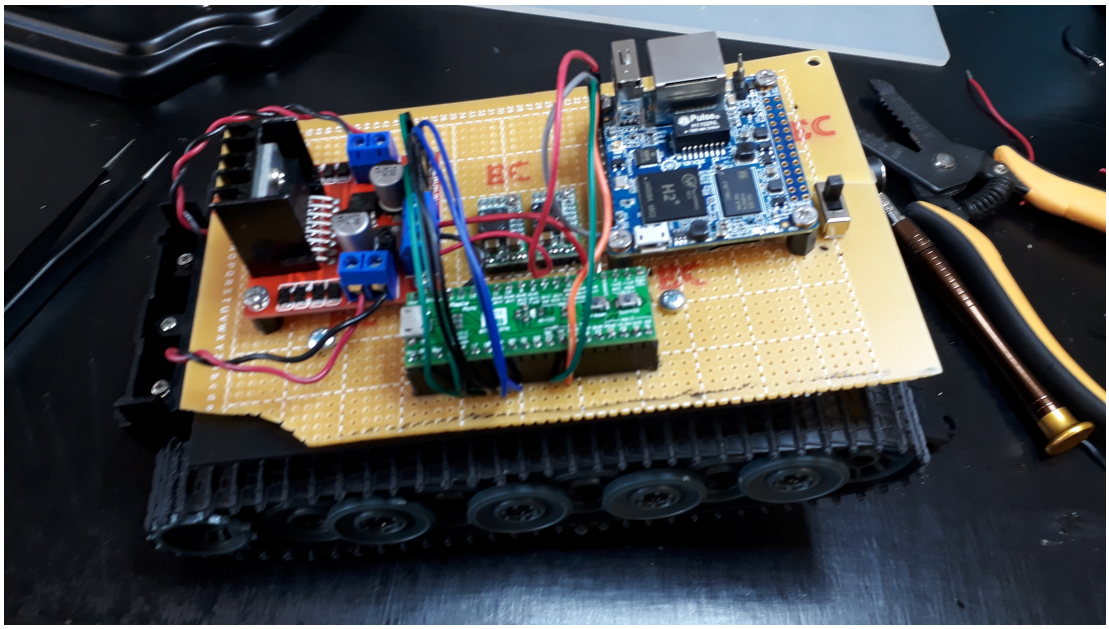

Figure 7. The completed assembly of the tank

The completed electrical schematic is shown in the appendix 5 "Electrical schematic of the system". In this schematic is the display of the general overlay of the system as well as

the general connections between components. The batteries themselves are connected to a battery board with three wires, positive, negative and a measuring wire, that comes from in between the two batteries. From the battery charger there are connections through a Molex plug to the DC plug for external power connections for battery charging and connection through a switch to the DCDC converters.

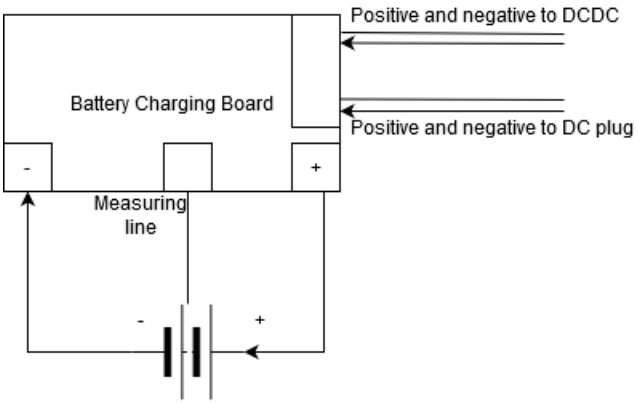

Figure 8. Battery connections

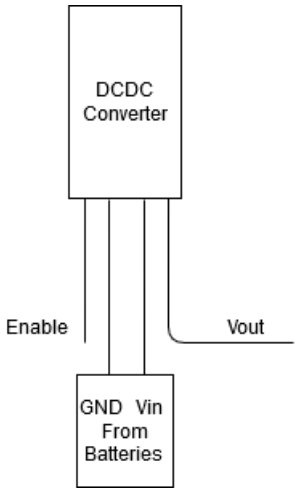

Figure 9. DCDC converters inputs and outputs connections

From the first DCDC converter go connections to the H-bridge, microcontroller board and Linux module, providing the 5 volt power into their power pins. The second DCDC converter is connected only to the H-bridge, providing a 7-12 volt voltage to supply the motors with enough power to run them. The converers themselves have 4 pins total, input voltage, output voltage, ground and enable, which in our case is not used.

Between the Pi Zero and Maple mini is only a connection between the USB pins so that they can communicate between each other. From the STM32 board go six wires to the H-Bridge. Two of them give PWM signals to the H-bridge to use to control the speed of the motors. Four more wires are used to control the direction of the motors, allowing for

movement in either direction, free movement of the motors or fast stopping depending on the input given. From the Maple Mini there are also connections to other, smaller components, like LEDs to show the turning on and off of the lights. The LED's are connected to 4 and 5 pins on the Maple Mini board and PortA 7 and 6 ports, respectively, on the STM32 Microcontroller.

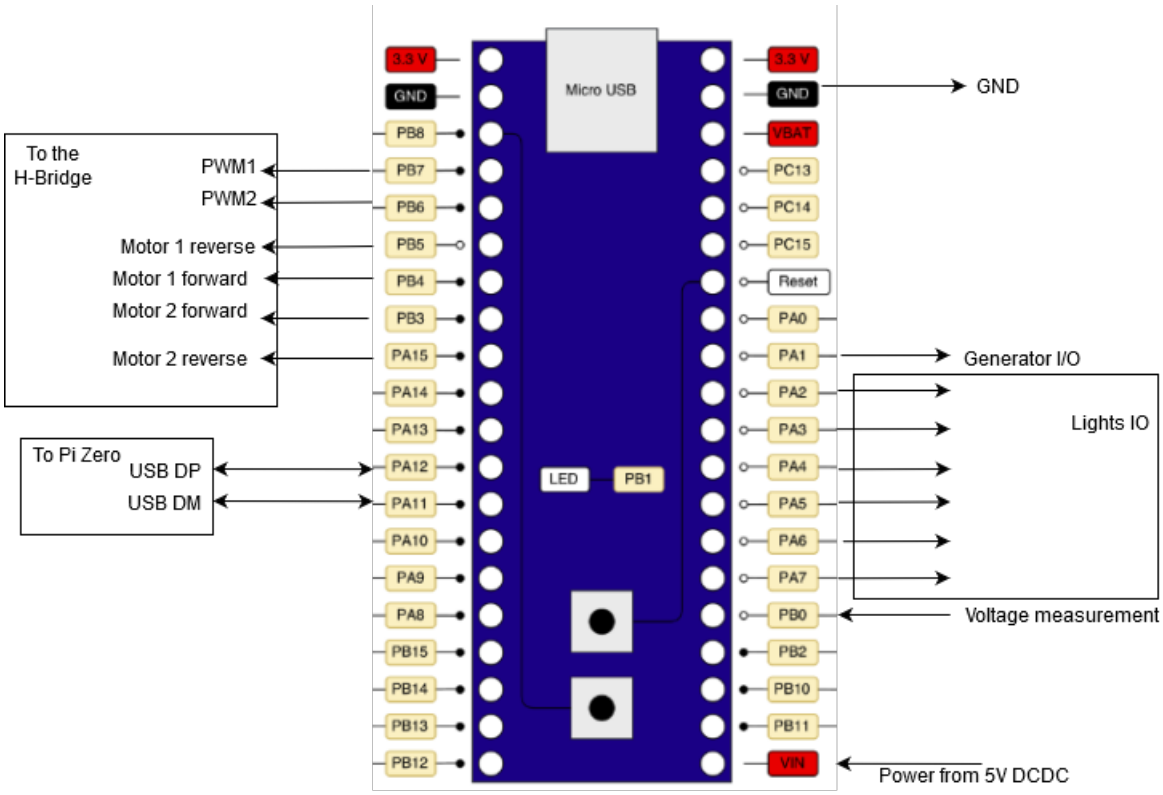

Figure 10. Maple Mini connections

With the motor to H-bridge connections it was a simple matter of connecting the positive and negative wires to the correct output on the bridge. There was a need, however, to make new wiring for the motors, due to the original wires breaking off the factory-made connections.

1.4 Problems with assembly

There were quite a few problems that arose in the course of the assembly and after the fact. When assembling the tank, there were multiple modifications that had to be made to accommodate the components. These were discussed in the previous sections of this chapter.

During the first testing of the assembly, there were a few things that got broken. There were three components that burned up or broke: the Maple Mini microcontroller, the Orange Pi Zero and one of the DCDC converters. All of these things were destroyed in the first testing and in the aftermath, when individual components were being tested. All of these parts had to be replaced. The broken parts were unsoldered from the prototyping board. They were also tested by trying to give them power and seeing if they worked or not. With the DCDC converter the on-board cover of the wire winding had broken and the Maple Mini and Orange Pi Zero would not work and caused power fluctuations when connected to other components. The new components were each soldered, if needed, back to the prototyping board, after having been tested separately.

There was also another instance, where the microcontroller was broken. During testing a few connections with the LEDs put onto the prototyping board, a connection broke free and shorted the LED lights and the Maple Mini. The symptoms in this case were the LEDS not working and the Maple Mini's inputs being connected straight to the power input. Because of this, these components had to be replaced and re-soldered.

Another issue that arose was during the first programming sessions. That was the issue of improperly made connections, as when trying to test the motor control, the motors would only run when the controller was a little bit out of it's socket and one of the wires, responsible for delivering the forward signal, was in a specific position. The wiring was unsoldered from the board and new wires were chosen and soldered onto the board. One of the reasons for new wires was that now they could be redone with enough length to make sure they reached between components and there would not be unnecessary pulling or stress on the wires.

However, some wiring issues came out much later in the testing and programming sessions. For example, the USB connection between the microcontroller and Linux module also proved to be an issue. While testing the connection between the two separately from the rest of the system, the connection seemed to be fine, although a few miniscule problems persisted. However when the whole thing was put together, then the STM32 and Pi Zero could not get a connection via the USB ports. To fix this new wires were chosen and

soldered to the prototyping board.

There was also an issue with the LEDs as one set kept burning out. As it turned out, the initial connections were accidentally made to the 5V pins, thus breaking the LEDs. After switching to the 3.3V outputs and connecting the LEDs with the proper resistors, the issue was fixed. As the LEDs used in this case were red and yellow, the voltages coming through were also required to be lowered, 1.8 and 2.4V respectively. The amperage going through them in this case was around 20mA and thus the resistors used for each LED were around 75 Ohms for the reds and 47 Ohms for yellows.

2. Programming of the system

In this chapter the programming portion of the thesis work is discussed. The portions that are discussed are about the chosen IDE for main programming, the programming of the Maple Mini and main controlling portion and the communication programming in the Orange Pi Zero Linux module.

The already existing solution is written in the same languages as the current one, mainly in C and Python. However, the main controller code is rather simplistic, allowing for only motor control. Said codes are also written with the ATmega328 microcontroller in mind and use Arduino's own libraries for main programming.

2.1 Programming IDE

This chapter covers the programming of the system. As part of the programming process, multiple IDE's were tried and used. The choices included Arduino based IDE, which was quickly opted out of, since it was a rather bare-bones and did not allow for good enough programming.

Early in the development mainly the STM32CubeIDE was used, which is the official ST Microelectronics programming IDE for STM type controller boards. The programming environment itself is free to use and provides multiple simplifications for programming the controllers. The environment also provides the way to flash and debug the STM32 board and to do an easy setup for the controllers input and output pins, timers and other internal components. In the early stage the IDE provided quite a lot of help and tips to work with, including writing and compiling a lot of code by itself, thus allowing for easy development. However, when it came to programming the communication between the STM32 board and the Linux module, the programming IDE became a blocker.

The problem in this case arose when we tried to create a new custom communication interface. To communicate between the Pi Zero and STM32 an custom USB interface was required that would accustom the custom USB2CAN library and descriptors that we used. As this was a custom interface, it required changes made in the STM32 auto-generated code, however, as the work continued, it became apparent that a lot of changes and testing

would be required. After some time, it was decided that working with the STM32CubeIDE would require a lot of time and, as the tests done so far were unsuccessful, another IDE and method would be used.

In the end the CLion IDE from Jetbrains toolbox was used, along with custom STM32 libraries made for use in other programming environments. The specific library used is called Libopencm3, which is an unofficial library for many micocontrollers, including the STM32F103C6T8 controller we are using. The CLion environment was chosen, as there was already access to it and it was the most familiar to the author. With the library used, the communication problems were eventually fixed and work could continue.

2.2 Programming of the STM32

In the early part of the development, the main idea was to get the parts working on their own first and then combining them. The different parts were the motor control, communications, input/output and others. With every part there were some issues, which will be covered in their own section below.

The general flow of the controller code would be to first to the necessary initializations, setting up the clocks, the required GPIO ports and the timer. After that the USB could be engaged and the USB polling could begin. During the polling, regular checks for inbound messages would be done. If received, certain actions would be taken, ending with the received values also being delivered back via the USB connection.

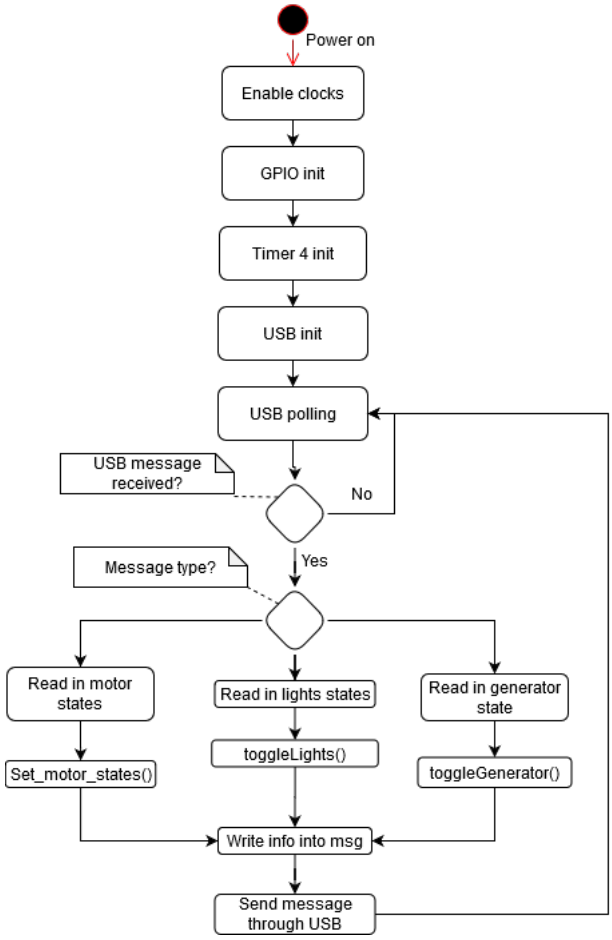

Figure 11. Main algorithm of the system

2.2.1 Controlling the motors

To control the motors a few changes had to be made regarding the I/O pins on the STM32 board. The pins first had to be configured properly. Two of these pins were set to be PWM pins and four were set to be output pins. The latter were connected to the H-Bridge's direction pins.

In order to control the motors directions, 4 output pins were used in total. Two of them would determine, if the motors would move in one direction and the other two would determine the opposite direction. The reason behind this is the fact that on the H-bridge board there are 4 pins for direction control as well and thus 4 pins needed to be used on the microcontroller.

In this case, the pins we used are labelled pins 17, 18, 19 and 20, which respectively correspond to the STM32's portB's outputs 5, 4 and 3 and portA's output 15. The control they provided is also accordingly motor 1's reverse and forward commands(portB 5 and 4) and motor 2's forward and reverse commands(portB 3 and portA13). Due to the motors

actual positioning and connections there had to be some testing done to determine, in which case the motors rotated in the same direction. All of these were also configured with an output mode of 2MHz, as it was fast enough for simple I/O.

Two more pins that were configured for motor control were the PWM pins. For these the onboard pins, which were used, were pins 15 and 16, which correspond to the STM32's portB outputs 7 and 6. These ports were chosen as they are able to provide the PWM signals needed to control the motors, as well as for their placement on the board, next to enough I/O pins to have their own "section" on the board. These were configured with the output of 50MHz as they needed to be updated more often than GPIOs. The pin configurations in the code are shown in the appendix 1.

For PWM creation we used the STM32 microcontrollers timer number 4 and its channels 1 and 2. These correspond to the ports and pins discussed and setup prior. In our case, the numerical values chosen were chosen, as they suited the best from the testing we contucted. The pin configurations in the code are show in the appendix 1.

The first things for the timer initialization were to reset the timer peripheral and then set its global mode. For the mode set, the timer was set to use no clock division ratio, edge based center-aligned mode and to count the values up. Then the prescaler was set, in our case to 72, so it would easily divide with the clock speed. For other setups the pre-load for the timer was disabled and then the timer was set to continuous mode. Finally in this portion, the timer's period was set, which we set as the maximum input from the operator (value 255) and multiplied it by 10. Lastly for the overall timer, various break and deadtime values were configured.

Next the timer channels were configured, first by disabling and clearing the outputs and then setting the preload. Next the channels were set to slow mode and to use PWM1 configuration. Channels and their N values were set to use high polarity and idle state. After setting the channel compare values to zero, to make sure that no preemptive movement happened, the timer channel's outputs were enabled, and the timer was enabled as well. The timer initialization code is show in the appendix 2.

To control the motors, the two functions were used, one for each motor. Simply put, the functions were called out in the CAN communication with inputs set from the operator messages. From said values the speed, or in our case PWM, value was sent to the respective timer channel and another value determined the motors move direction, if a direction was set. The former was done using the same function call as in the timer initialization and the latter using GPIO set and clear functions.

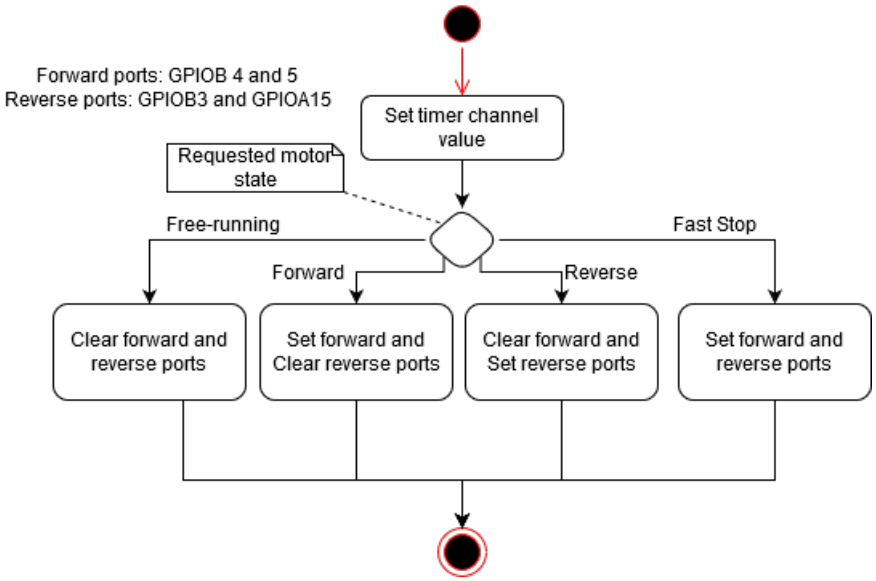

Figure 12. Flowchart for motor controlling function

2.2.2 Lights and Generator I/O configuration

For the lights we reserved the pins 4 to 9 on the Maple Mini board. These were chosen as they provide an 3.3V output, allowing for better suited power output to the LEDs. The corresponding ports were portA 2 to 7 on the STM32 chip. These outputs were set to 2MHz as well as they do not need much faster input. These ports are controlled through the CAN messages discussed in another section, and are simple turn on/off commands.

The lights control was done by taking the input from CAN, inputting the values into an array, and feeding the array into the toggleLights function. In said function, each of the values is checked and the specific pin is either set high or low accordingly. Do note that one of the lights is currently left out of the code, as it was not inputted and, in reality, with other lights, besides the front and rear work lights, no physical output is displayed either.

For the generator simulation a pin and port were reserved as well. For this the pin10 was chosen on the board, which corresponds to portA 1. Just as with the lights, the output is set currently to 2MHz, done through CAN messages, and controlled by a simple on/off. In the future versions this should be determined by vehicle modes. The input is taken from the CAN communications as well and at the moment is a basic input/output switching, just as with the lights. The pin configurations in the code are show in the appendix 1.

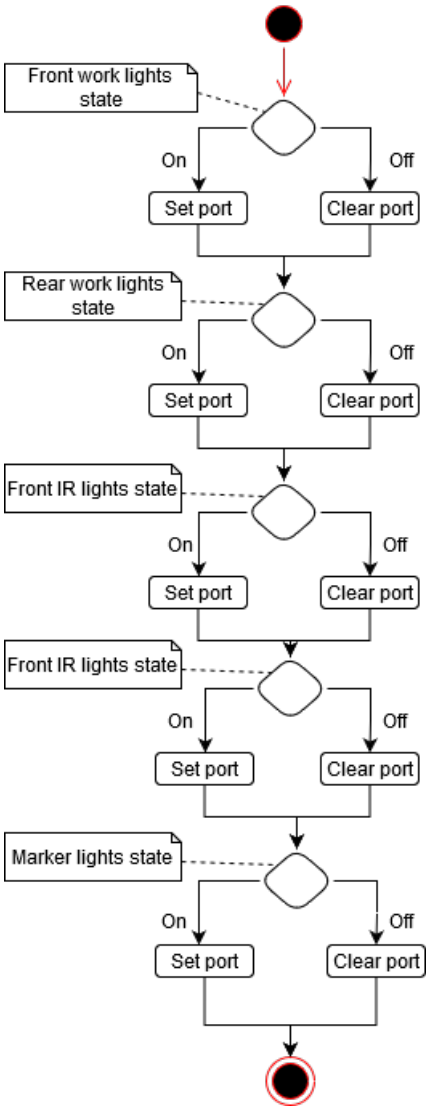

Figure 13. Flowchart for light input/output

2.2.3 Battery charge checking

The battery charge checking portion is rather simple. On the Maple Mini board we used pin 3, which corresponds to portB 0 in the STM32's configuration. This pin allows the measuring of analog signals and as such is used to measure the input from the battery management board, or from power input. The pin configurations in the code are show in the appendix 1.

As for the measurement, since the input is in analog and the pin allows for max voltage of 3.3V, the pin pin was connected to with two resistors, taking the voltage down to a readable level. To get the real value, we calculated the original value in the function we read the value in with the reference voltage of 8.4V, which is around the max charge for the batteries. This value would go onto the UDP communication to be sent to the operator.

```
float read_adc() {
  int input = gpio_get(GPIOB, GPIO0); // Read the input from
            //the analog pin
  float inputVoltage = (((float)input / 4096) * 8.4);
  // Calculate the value against the reference voltage
  return inputVoltage;
```

Listing 2.1. Analog port reading

2.2.4 Communication

In order to simulate the CAN communication between the STM32 and Linux, connections were made on the pins 23 and 24, corresponding to portA 12 and 11. However, there was no real configuration needed for the pins exactly. Instead, once the USB peripheral clock was set, the USB connection itself could be made. However, this did require a configuration of another port, which would have to be set to ON for the configuration and kept on, until the CAN/USB communication was set. The port in question was portB 9, which does not have an physical pin, but is used in USB communications.

```
/* !
 * ! USB Triggering
 * ! A thing with Maple Mini where an output is required to
 * ! be toggled in order to send and receive the initial
 * ! setup via USB.
 * ! PB9
 */
  gpio_set_mode(GPIOB, GPIO_MODE_OUTPUT_2_MHZ,
                GPIO_CNF_OUTPUT_PUSHPULL, GPIO9);
  gpio_set(GPIOB, GPIO9);
```

Listing 2.2. USB trigger setup

In addition, we also had to set up custom USB descriptors following the USB descriptor standard to make sure the Linux module would recognize our microcontroller. This required setting up the general USB2CAN interface, configuration and endpoint descriptors. The most relevant piece here would probably the endpoint descriptors configuration, as it was needed to set up four endpoints. Two of these were for command endpoints and two for general receive and sending endpoints.

After the descriptors were set up, a small command/configuration message was made to be sent through the connection and after that the real communication could happen. In the following part the data that was sent was either converted to or from the USB message into a readable for us CAN message. This was done on both the Linux and STM32 side.

```
static void usbdev_cmd_rx_cb(usbd_device *usbd_dev, uint8_t ep) {
  struct usb_8dev_cmd_msg incmd;
  struct usb_8dev_cmd_msg outcmd;

  (void)ep;
  (void)usbd_dev;

  outcmd.begin = USB_8DEV_CMD_START;
  outcmd.end = USB_8DEV_CMD_END;

  int len = usbd_ep_read_packet(usbd_dev, USB_ENDPOINT_ADDR_OUT(4), &
   incmd, 64);
  if (len) {
    usbd_ep_write_packet(usbd_dev, USB_ENDPOINT_ADDR_IN(3), &outcmd,
   len);
  }

  static void usbdev_set_config(usbd_device *usbd_dev, uint16_t wValue)
    {
  (void)wValue;
  (void)usbd_dev;

  usbd_ep_setup(usbd_dev, USB_ENDPOINT_ADDR_IN(1),
   USB_ENDPOINT_ATTR_BULK, 64, NULL);
  usbd_ep_setup(usbd_dev, USB_ENDPOINT_ADDR_OUT(2),
   USB_ENDPOINT_ATTR_BULK, 64, usbdev_data_rx_cb);
  usbd_ep_setup(usbd_dev, USB_ENDPOINT_ADDR_IN(3),
   USB_ENDPOINT_ATTR_BULK, 64, NULL);
  usbd_ep_setup(usbd_dev, USB_ENDPOINT_ADDR_OUT(4),
   USB_ENDPOINT_ATTR_BULK, 64, usbdev_cmd_rx_cb);
}
```

Listing 2.3. USB2CAN command receive and sending configuration

In general, the info was then analyzed and then appropriate actions were done with said data. In our case there were three main message that were sent, although initially only two were actually done. One message would be for the motor control, one would be for the lights control and one would be for generator simulation control(buzzer control). Although the last was not actually used, it was done for future use. The main code is shown in appendix 3 "Motor control code".

The first message, motor control, is sent with a CAN ID of 0xA in hexadecimal and consists of 4 bytes of data. The first two bytes say, which direction each motor should move(forward or backward) and the other two bytes are used for PWM control. While the states are in the range of 0-3(the value definitions are discussed in section motor control),

the PWM value can range from 0 to 255(8 bit value). For our use, we multiply the value by 10 to give better control over the PWM values sent out to the motors.

```
else if (tx_msg.id = 0xB) {
      uint8_t lightsState[5] = {tx_msg.data[0], tx_msg.data[1],
                                tx_msg.data[2], tx_msg.data[3],
                                tx_msg.data[4]};
      toggleLights(lightsState);

      rx_msg.dlc = 5;
      rx_msg.data[0] = tx_msg.data[0];
      rx_msg.data[1] = tx_msg.data[1];
      rx_msg.data[2] = tx_msg.data[2];
      rx_msg.data[3] = tx_msg.data[3];
      rx_msg.data[4] = tx_msg.data[4];
    }
```

Listing 2.4. Lights message decoding

The second message is the lights message. With a CAN ID of 0xB in hexadecimal, it's message consists of 5 bytes of data. Each of these bytes controls a different set of lights. Starting from the beginning, the lights controlled are: front work lights, rear work lights, front IR lights, rear IR lights, marker lights, and combat lights. Of those only the work lights are currently implemented, due to time restrictions and others will be added in the future iterations of the mini-tank.

```
    else if (tx_msg.id = 0xC) {
      uint8_t genState = tx_msg.data[0];
      toggleGenerator(genState);
      rx_msg.data[0] = tx_msg.data[0];
    } else {
      rx_msg.dlc = 8;
      rx_msg.data[0] = tx_msg.data[0];
      rx_msg.data[1] = tx_msg.data[1];
      rx_msg.data[2] = tx_msg.data[2];
      rx_msg.data[3] = tx_msg.data[3];
      rx_msg.data[4] = tx_msg.data[4];
      rx_msg.data[7] = 0xFE;
    }
```

Listing 2.5. Generator message decoding and error creating.

The last currently implemented message is for the generator. In this case, the CAN ID is 0xC in hexadecimal and it only consists of one byte, which is the desired generator state. This is essentially only a ON/OFF signal, which determines if our buzzer, which is essentially our simulation of the generator, is working.

For each of these signals the values and data received are also sent back to the Linux module and from there to the operator. The only exception is if, for some reason, the data received is not one of our packets. In that case the info sent back is the same as received, with the exception of an extra byte, hard-coded to the value in hexadecimal of 0xFE, differentiate it from others. This essentially shows that the CAN message received is not one that we can work with.

2.3 Linux module

For the Orange Pi Zero, the first course of action was to choose a Linux distribution to use with the system and install it on a micro-SD card. The chosen system in this case is Armbian OS, which is small enough, and officially supported, for our system both in capabilities and in size of the operating system. This OS also provides enough support on its own to do the initial setup and modifications to the system. The modifications mainly concern the SSH and WIFI connections as well as setting up the CAN2USB connection.

To configure and set up the Pi Zero, we used SSH for regular modification for both system wise and code changes. We also used SFTP connection for file transfer, as a few files had to be made and added to Linux. The main files being the Python code that runs the operator-to-STM32 connection.

Said code is rather simple on a broader scale. The Python code sets up the network UDP connection using the Linux libraries, most notably the Linux Socket libraries, and on it's own starts sending out messages waiting for connection to the operator. The messages are broadcasted out with the operators IP and the operator side will listen for any messages coming its way.

The code will also set up a CAN connection. It will look for an CAN interface and, if it exists, connect to it. If it finds the interface, it at first will start to send empty messages to make sure that the connection is kept up. The messages are essentially empty motor control messages, with the ID 0xA in hexadecimal and all the data bytes set to 0. Only once the operator has confirmed the connection on their side, do the rest of the message start being sent. As a part of all this, the code also does the conversion of messages from UDP to CAN and vice-versa.

A lot of the Python code used in the project could be based off of the code used for the previous version of the tank. Because the connection side between the operator and the system was rather similar, there were not that many changes that had to be made. The main changes involved around the connection creation, the need to change where and how

the system connect via the WIFI and UDP connections, and the CAN message creation and dissection for the UDP message. For the latter we had to figure out the method to create new messages and fields for sending the information to the STM32 microcontroller. The end result was not the hardest to do and this portion of the code was rather quickly finished. As with other bigger codes, this code can be seen in the appendix 4.

2.4 Problems with programming

One of the first problems with the programming was choosing out the IDE to use. Since there were a lot of choices, multiple programming environments were tried. With each came different positives and negatives: some had better options for bootloading the board, others had better support for actual programming.

Some problems arose with the bootloading as well, as the the first tests with bootloading and using the on-board HAL were rather unsuccessful. The issue in this case was due to the use of a bootloader program called DFU-Util(Device Firmware Upgrade Utilities). The program did not flash the code onto the board properly and thus the microcontroller's HAL did not execute properly. To fix the issue, the method of flashing the board was switched to using an ST-Link V2 JTAG flasher. This change worked and the issue was fixed.

With the STMicroelectronics STMCubeIDE the initial code writing was a success as it allowed for rather good support, specially with a GUI allowing for easy setup of the GPIOs and PWM ports and other parameters. The issues with the IDE arose when trying to work out custom solutions with it. For one, the ST-Link JTAG connection used to flash onto the STM32F103 chip would sometimes not work, which essentially required a redo of the environments flashing setup. Another issue was already talked about at the beginning of the chapter under the "Programming IDE" section. To summarize, even though STM allows for custom USB descriptors, they do still follow a strict setup, which in our case was difficult to do. Thus the IDE was abandoned and another custom solution was found.

With timer the issue was mainly trying to get the correct set up done, with the correct parameters. The first tests were done by trying out random values, later tests were done by using a few calculations based on general PWM calculations.[6] Final values are, however, based on the information taken from the libopencm3 libraries own example Git repositories.[7] Although this code is made for another microcontroller, the general setup is similar to ours and thus it was used to get a general setup done using our controller's library. The end values were then chosen through some testing.

There was also a need to set up the CAN network for the Linux module every time the

USB/CAN connection was made. This was due to Linux not setting up the connection itself and the user needing to do it manually. This was initially done with Linux terminal commands. In the end, a rc.local file was created in the "root/etc" folder, which would run every time the Linux module would boot up. The file was written in bash and essentially continuously checks, if the CAN connection is available, and initializes and sets up the network, if it exists. Otherwise, it would display an error in the kernel's journal. This streamlined the start up process significantly and removed the need for the user to input the commands.

```
#!/bin/bash
# rc.local
intUp=0
while [ $intUp -eq 0 ] # Until the can network is not up
do
        sudo ip link set can0 type can bitrate 500000 && sudo ip link set can0 up qlen 1000 # Try to set up the CAN network
        if ip a show can0 up | grep -n 'state UP' #if it's up
        then
                echo "CAN0 Up"
                intUp=1;
        fi
done
python3 /home/tankipi/TankPPy/pytank/tank.py # Run the communication script
exit 0
```

Listing 2.6. The bash script for setting up CAN

3. Testing of the system

This chapter covers the various stages of testing. This includes testing on an individual component or unit level and testing with the system as a whole. The main parts of the system that needed serious testing were the robots, and thus motors, movement, the communication and input/output switching, mainly for the LEDs.

3.1 The operators interface

For testing purposes the operator interface we used was the same that is used on the real UGVs. The reason for the use of this interface are the desires to use the system as a learning tool for new users as well as for overall, easier, testing. Not to mention, as the very final result should have the actual UGVs code running on it, making sure that the operator interface is usable with this system is a definite requirement. For this interface no modifications were made as it was used just as a tool to test out the system and to make sure the connections and the system as a whole works.

The operator interface consists of two parts: the controller and the operator UI. The controller used for testing in this case was an XBox style controller. The reason for the use of said controller is that said controller gives the most consistent layout and is able to connect to most of the operating systems out there. It is also the type of controller that the indoor testing is usually done with.

As for the operator UI, there is not much that can be openly said. As with most non-field testing, the operator UI ran on a laptop, where it could be accessed with the mouse cursor. With said UI, the main things that were done, were the checking to make sure that the connection was still up, as well as the lights I/O and generator switching on and off.

3.2 Testing process

The testing process can be divided into two parts: component level testing and full system testing. The methods used and tests done are discussed in the following sections.

3.2.1 Component level testing

The very first tests that were done, were done at the beginning of the project. They were simple unit tests, to check, if the components work on their own. The general test plan for the component level was to simply give them the necessary inputs and then see, if the results are what we expected.

During this portion the motors and H-bridge were tested to see if they work, by giving them power and inputs in the correct pins. The STM32 microcontroller was tested multiple times to see if the input and outputs are working and, of course, if the wire connections worked. This was also the main portion of the testing, where various mistakes happened that lead to the destruction of quite a few of the components.

Initially there were a lot of issues with the wire connections made and thus there were issues getting motors to work. After rewiring, it was needed to set up the PWM timers, which required quite a bit of testing to get right. Some testing was also required to get the direction pins correctly set up.

Another of the initial tests was testing of the connections on the Orange Pi Zero. This was done by setting up a physical connection between the Maple Mini board and the Pi Zero. This was done to see if the USB2CAN could be set up and that the messages moved between the components. Initially there were issues with the setup, as on the Linux the connection required to be set up each time the physical connection was made. For some reason, in the beginning, the messages would also sometimes not transmit, most likely due to a conflict with Linux libraries.

3.2.2 Full system testing

The general test plan for the full system was as follows:

- Turn on the system. Begins by flicking the switch to the ON position. All the components on should turn on and this portion of the test should end by the tank appearing on the operator UI.
- Connecting to the robot, by setting the initial values from the UI.
- Once that is done, simple movement tests are made. The robot is moved forward, reverse, turned left and right and forward-left, forward-right and both ways in reverse.
- Then the I/O is tested. The lights are turned on and off and generator pin is turned high or low, if possible.

Once these tests are done, the robot is working as expected and can be moved around, as long as it stays within the WIFI network. Do note that this testing plan only applies, in the most part, to the portion, where the Orange Pi Zero was connected. If it was not, the free movement of the robot could not be tested, as is said in the following paragraphs.

The first post-assembly testing was not done with the Linux module, as it was deemed easier to use a computer with a Linux distribution on it. There were a couple of modification that had to be made in the Python script to use it in this case, but it would allow for easier modifications of the code if need be. However the the testing could not be done fully in this case, as it required an USB cable connection between the computer and Maple Mini board. Thus the mini-tank could not move freely and testing had to be done holding or on a platform.

During this testing the motor control and lights I/O was tested. The motor tests were done by moving the Xbox controllers joysticks and seeing if the motors move accordingly. For the most part, the motors worked correctly, but there an issue arose with said controller and the Linux script, this will be talked in the problems subsection. With I/O the testing was done through the operator UI. By pressing button on the UI, the messages were sent out to the Linux module and, through that, the STM32. For the I/O there seemed to be no issues that arose.

The next testing was done with the Linux module connected. During this testing one last connection issue arose, as when the USB connections were made between the Linux module and the Maple Mini, Linux could not detect the connection. After fixing the connection the CAN interface appeared on the Linux module and could be tested.

The process in this case was similar with the testing done while connected to a computer. The main difference in this case was that, with the Linux module connected, the robot could move freely around. The Linux was connected to a local WIFI network, through which the messages were sent with the UDP connection.

The movement and I/O switching was done in the same manner as before, with the exception of, as said before, the robot being able to move freely. As long as the UGV stayed within the reach of the nearest WIFI node or router, the robot could be controlled.

3.2.3 Problems during testing

One of the first issues that arose was the problems with various connections on and between the boards. Quite a few times changes had to be made, because one or more connections,

in this case mostly wires, happened to be loose or completely broken. In addition, there were quite a few mistakes made in the early stages, most of which have already been talked about multiple times.

Best examples of this would be the situations where a wire would come open during testing and short-circuit another part of the board. In one case the wire that came open was even a power block connector, which was used to check the status of the LEDs. In this situation, all of the LEDs that were connected had to be replaced as well as the Maple Mini board. The latter had been short-circuited and did not work properly anymore, giving out the full outputs to every pin.

Another example of this case is towards the very beginning, after the initial assembly of the tank. While measuring the voltages across the board, one of the multimeter's prongs slipped out of the author's hand and as a result multiple components had to be replaced. This includes the Maple Mini board, once again, the Orange Pi Zero and the DCDC converters. While the former and the latter simply did not work, and one of the DCDC converters being physically broken around the winding, the Pi Zero caused an intense drop in voltages any time it was connected.

During the early full system testing there was also an issue with the chosen joystick controller. The initial testing was done with an third-party controller. Although similar to the XBox one used later on, it caused an issue with the joystick controls, where the values received from the controller would underflow and go back into positive values in certain position. This would cause the program receiving the inputs to display an error and quit the execution. The issue was with said third-party controller and afterwards, when the official XBox controller was used, this issue did not come up.

3.3 Improvements and additions

During the testing mainly improvements of physical nature were made. Throughout the testing, various connections were redone, re-soldered, and placed around. In addition, after the first tests a few components were added, like the LEDs and buzzer. There were also improvements and changes made in the code, as motors and USB2CAN connections were tested. Most of the improvements made during tested are currently present in the code as well and discussed in the previous chapters.

As the result of this work was an general base robot, upon which the future iterations could be made, there are quite a number improvements that can be made. The first improvement that can be done, would be the implementation of more inputs and outputs to simulate

other parts of the full-sized UGV. There are multiple parts of the system, that can be simulated with various components, like displaying the status of battery system and *et cetera*. These could be done by a number of ways, for example displaying a LED to signify if the subsystem is turned on or off, among many other possibilities.

Another part of the system that can be built upon is the communication. Currently the communication itself is rather simple and allows only for certain types of messages to be sent and analyzed. If this part of the system was continued as is, it would become quite a large code block. Thus this portion of the code would have to be redone, if more components are added.

A camera system could also be added. On the first version of the robot, which was the basis for this one, a camera possibility was already added. In this work, this was not planned and thus could be added back on in an future iteration. However, this camera system would be worked onto the Orange Pi Zero instead of the microcontroller, as the latter would have difficulties processing the video feed and the sending of it to the Linux module and operator. The feed from this camera would be displayed on the operator's UI and quite possibly be used to do various tasks in regards to the image processing.

If we wanted to implement the UGV's code into the system, there would have to be even more changes made to the code. As the main code itself runs on Linux already, there would have to be modifications made to complement the code on the current board. Physically little changes would have to be made, but software wise, there would have to be a number of changes made. In addition to the changes mentioned prior, which would be made mostly on the STM32's side, there would most likely be changes on the codes side as well, since there would be differences between the actual system and the one done in this work.

4. Summary

The task of this thesis was to create a small scale mock-up of a real life UGV that could mimic the movements and basic I/O switching and be the basis for future improved versions of similar nature. Said system had to use various components to create a physical platform to be used and programmed to have a similar behaviour as the real thing.

The end result is a mini-robot that copies the basic movement and I/O of said UGV and is controlled in the same way through an operator UI and operator controller allowing for similar base level feedback to the user as the real thing. Said robot uses various components, like the Maple Mini with the STM32F103 chip, Orange Pi Zero and more, that were assembled to create the full-system. The end product was programmed in multiple languages, the main language being C and with other parts being in Python and a bit of Bash scripting language.

The system was tested a few times. In particular, the system's movement, communication and I/O switching were the main concerns, as they had to mimic the real UGV's actions as closely as possible. The robot can move in all directions and turn on or off the multiple I/Os on it. During the testing, there were some issues with the system that arose and improvements were determined that could be made in the future. The end result, however, was deemed acceptable.

With the end result it is possible to provide support for smaller programming testing as well as helping new users learn about the full system. To give better support there needs to be improvements and additions made with the end result being as close to the real UGV as possible.

Bibliography

[1] ST Microelectronics. *Mainstream Performance line, ARM Cortex-M3 MCU with 64 Kbytes Flash, 72 MHz CPU, motor control, USB and CAN*. [Accessed: 23-03-2020]. 2018. URL: https://www.st.com/en/microcontrollers-microprocessors/stm32f103c8.html.

[2] STM32-Base. *Baite Maple Mini Clone STM32F103C8T6*. [Accessed: 23-03-2020]. URL: https://stm32-base.org/boards/STM32F103C8T6-Baite-Maple-Mini-Clone#USB-connector.

[3] Xunlong Software CO. *What's Orange Pi Zero?* [Accessed: 01-04-2020]. URL: http://www.orangepi.org/orangepizero/.

[4] *Votlage regulator step-down 3A*. [Accessed: 05-04-2020]. URL: https://robolabor.ee/en/converters/976-votlage-regulator-step-down-3a.html.

[5] Last Minute Engineers. *Interface L298N DC Motor Driver Module with Arduino*. [Accessed: 05-04-2020]. URL: https://lastminuteengineers.com/l298n-dc-stepper-driver-arduino-tutorial/#l298n-motor-driver-module-pinout.

[6] Engineers Garage. *Stm32f103 Pwm(Pulse width modulation) signal generation using internal timers, keil MDK-ARMv6 and Stmcubemx Ide*. [Accessed: 10-04-2020. 2019. URL: https://www.engineersgarage.com/stm32/stm32-pwm-generation-using-timers/.

[7] Fabian Inostroza. *Libopencm3-examples*. [Accessed: 05-04-2020]. 2018. URL: https://github.com/FabianInostroza/libopencm3-examples/tree/master/examples/stm32/f4/stm32f4-discovery/pwm.

Appendices

Appendix 1 - General Purpose input and output initialization code

```
static void gpio_init() {
/* !
 * ! Timer setting
 * ! PB6 - Right motor PWM
 * ! PB7 - Left motor PWM
 * ! GPIO pins are set to 50 MHz and use alternate function(Timer PWM)
 */
  gpio_set_mode(GPIOB, GPIO_MODE_OUTPUT_50_MHZ,
      GPIO_CNF_OUTPUT_ALTFN_PUSHPULL, GPIO6 | GPIO7);

/* !
 * ! LED PIN
 * ! Mainly for use for debugging.
 * ! Also shows that the board works
 * ! PB1, 2MHz
 */
  gpio_set_mode(GPIOB, GPIO_MODE_OUTPUT_2_MHZ,
                GPIO_CNF_OUTPUT_PUSHPULL, GPIO1);
  gpio_set(GPIOB, GPIO1);

/* !
 * ! Motor control pins
 * ! Outputs to control the motor directions.
 * ! PB5 - Left Reverse (Pin 17)
 * ! PB4 - Left Forward (Pin 18)
 * ! PB3 - Right Forward (Pin 19)
 * ! PA15 - Right Reverse (Pin 20)
 * ! Outputs are on 2MHz
 */
  gpio_set_mode(GPIOB, GPIO_MODE_OUTPUT_2_MHZ,
                GPIO_CNF_OUTPUT_PUSHPULL, GPIO3 | GPIO4 | GPIO5);
  gpio_set_mode(GPIOA, GPIO_MODE_OUTPUT_2_MHZ,
```

```
33                 GPIO_CNF_OUTPUT_PUSHPULL, GPIO15);
34   gpio_clear(GPIOB, GPIO3 | GPIO4 | GPIO5);
35   gpio_clear(GPIOA, GPIO15);
36 /* !
37  * ! Analog input pin setting
38  * ! For measuring the current fullness of the batteries
39  * ! PB0 - PIN3
40  */
41   gpio_set_mode(GPIOB, GPIO_MODE_INPUT,
42                 GPIO_CNF_INPUT_ANALOG, GPIO0);
43 /* !
44  * ! USB Triggering
45  * ! A thing with Maple Mini where an output is required to be toggled
      in order
46  * ! to send and receive via USB.
47  * ! PB9
48  */
49   gpio_set_mode(GPIOB, GPIO_MODE_OUTPUT_2_MHZ,
50                 GPIO_CNF_OUTPUT_PUSHPULL, GPIO9);
51   gpio_set(GPIOB, GPIO9);
52 /* !
53  * ! Light IO setup
54  * ! PA7 - PIN4 - Front Work lights
55  * ! PA6 - PIN5 - Rear Work Lights
56  * ! PA5 - PIN6 - Front IR lights(If added)
57  * ! PA4 - PIN7 - Rear IR lights(If added)
58  * ! PA3 - PIN8 - Marker Lights (If added)
59  * ! PA2 - PIN9 - Combat Lights (If added)
60  */
61   gpio_set_mode(GPIOA, GPIO_MODE_OUTPUT_2_MHZ,
62                 GPIO_CNF_OUTPUT_PUSHPULL, GPIO2 | GPIO3 | GPIO4 | GPIO5
        | GPIO6 | GPIO7);
63   gpio_clear(GPIOA, GPIO2 | GPIO3 | GPIO4 | GPIO5 | GPIO6 | GPIO7);
64
65 /* !
66  * ! Generator setup
67  * ! Analog/Digital output pin to turn on a buzzer to notify generator
      status
68  * ! PA0 - PIN11
69  */
70   gpio_set_mode(GPIOA, GPIO_MODE_OUTPUT_2_MHZ,
71                 GPIO_CNF_OUTPUT_PUSHPULL, GPIO10);
72   gpio_clear(GPIOB, GPIO10);
73 }
```

Appendix 2 - Timer initialization code

```
static void timer_init() {
  // Reset TIM4 peripheral
  rcc_periph_reset_pulse(RST_TIM4);

  // Timer global mode set
  timer_set_mode(TIM4, TIM_CR1_CKD_CK_INT, TIM_CR1_CMS_EDGE,
    TIM_CR1_DIR_UP);

  // Reset timer prescaler value
  timer_set_prescaler(TIM4, 72);

  // Disable preload
  timer_disable_preload(TIM4);

  // Continous mode
  timer_continuous_mode(TIM4);

  // Set Period
  // TIM4 Uses clock from APB2
  // Timer set period based on Operator input(0-255) * 10
  //
  timer_set_period(TIM4, 2550);

  /* Configure break and deadtime. */
  timer_set_deadtime(TIM4, 10);
  timer_set_enabled_off_state_in_idle_mode(TIM4);
  timer_set_enabled_off_state_in_run_mode(TIM4);
  timer_disable_break(TIM4);
  timer_set_break_polarity_high(TIM4);
  timer_disable_break_automatic_output(TIM4);
  timer_set_break_lock(TIM4, TIM_BDTR_LOCK_OFF);
  /* OC1 and OC2 configurations
   * Disable outputs. */
  timer_disable_oc_output(TIM4, TIM_OC1);
  timer_disable_oc_output(TIM4, TIM_OC1N);
  timer_disable_oc_output(TIM4, TIM_OC2);
  timer_disable_oc_output(TIM4, TIM_OC2N);
  /* Configure Global line */
  timer_disable_oc_clear(TIM4, TIM_OC1);
  timer_disable_oc_clear(TIM4, TIM_OC2);
```

```
  timer_enable_oc_preload(TIM4, TIM_OC1);
  timer_enable_oc_preload(TIM4, TIM_OC2);
  timer_set_oc_slow_mode(TIM4, TIM_OC1);
  timer_set_oc_slow_mode(TIM4, TIM_OC2);
  timer_set_oc_mode(TIM4, TIM_OC1, TIM_OCM_PWM1);
  timer_set_oc_mode(TIM4, TIM_OC2, TIM_OCM_PWM1);
  /* Configure the channels */
  timer_set_oc_polarity_high(TIM4, TIM_OC1);
  timer_set_oc_idle_state_set(TIM4, TIM_OC1);
  timer_set_oc_polarity_high(TIM4, TIM_OC2);
  timer_set_oc_idle_state_set(TIM4, TIM_OC2);
  /* Configure the Channels N values */
  timer_set_oc_polarity_high(TIM4, TIM_OC1N);
  timer_set_oc_idle_state_set(TIM4, TIM_OC1N);
  timer_set_oc_polarity_high(TIM4, TIM_OC2N);
  timer_set_oc_idle_state_set(TIM4, TIM_OC2N);
  /* Set the compare values to 0 */
  timer_set_oc_value(TIM4, TIM_OC1, 0);
  timer_set_oc_value(TIM4, TIM_OC2, 0);
  /* Enable the outputs */
  timer_enable_oc_output(TIM4, TIM_OC1);
  timer_enable_oc_output(TIM4, TIM_OC2);
  timer_enable_oc_output(TIM4, TIM_OC1N);
  timer_enable_oc_output(TIM4, TIM_OC2N);
  /* Enable timer preloading */
  timer_enable_preload(TIM4);
  /* Enable outputs in the break subsystem. */
  timer_enable_break_main_output(TIM4);
  /* Enable timer counter */
  timer_enable_counter(TIM4);
}
```

Appendix 3 - Motor control code

```
void set_motorA_state(uint8_t state, uint16_t pwm) {
  timer_set_oc_value(TIM4, TIM_OC1, pwm);
  switch(state) {
    case FREE_RUNNING:
      gpio_clear(GPIOB, GPIO4);
      gpio_clear(GPIOB, GPIO5);
      break;
    case FORWARD:
      gpio_set(GPIOB, GPIO4);
      gpio_clear(GPIOB, GPIO5);
      break;
    case REVERSE:
      gpio_clear(GPIOB, GPIO4);
      gpio_set(GPIOB, GPIO5);
      break;
    case FAST_STOP:
      gpio_set(GPIOB, GPIO4);
      gpio_set(GPIOB, GPIO5);
      break;
}}
void set_motorB_state(uint8_t state, uint16_t pwm) {
  timer_set_oc_value(TIM4, TIM_OC2, pwm);
  switch(state) {
    case FREE_RUNNING:
      gpio_clear(GPIOB, GPIO3);
      gpio_clear(GPIOA, GPIO15);
      break;
    case FORWARD:
      gpio_set(GPIOB, GPIO3);
      gpio_clear(GPIOA, GPIO15);
      break;
    case REVERSE:
      gpio_clear(GPIOB, GPIO3);
      gpio_set(GPIOA, GPIO15);
      break;
    case FAST_STOP:
      gpio_set(GPIOB, GPIO3);
      gpio_set(GPIOA, GPIO15);
      break;
}}
```

Appendix 4 - Python code for message sending

```
    if ret.sysCtl.frontWorkLights or ret.sysCtl.rearWorkLights or
                    ret.sysCtl.frontIRLights or ret.sysCtl.rearIRLights
    or
                    ret.sysCtl.markerLights:
                    front_lights = ret.sysCtl.frontWorkLights
                    rear_lights = ret.sysCtl.rearWorkLights
                    front_IR = ret.sysCtl.frontIRLights
                    rear_IR = ret.sysCtl.rearIRLights
                    marker_lights = ret.sysCtl.markerLights
                else:
                    front_lights = 0x0
                    rear_lights = 0x0
                    front_IR = 0x0
                    rear_IR = 0x0
                    marker_lights = 0x0
                b = array.array('B', [front_lights, rear_lights, 0x0, 0
    x0, 0x0]).tobytes()
                lights_state = build_can_frame(0xB, b)
                print('can_id=%x, can_dlc=%x, data=%s' %
    dissect_can_frame(lights_state))
                s.send(lights_state)
                # Generator(Buzzer) Switch on
                if ret.sysCtl.generatorOn:
                    generator = ret.sysCtl.generatorOn
                else:
                    generator = 0x0
                c = array.array('B', [generator]).tobytes()
                gen_state = build_can_frame(0xC, c)
                print('can_id=%x, can_dlc=%x, data=%s' %
    dissect_can_frame(gen_state))
                s.send(gen_state)
```

Appendix 5 - Electrical schematic of the system