NTNU
Norwegian University of
Science and Technology

Kybulf jr. The Walking Six-Legged Bug Robot

Edvin Holmseth

Master of Science in Cybernetics and Robotics
Submission date: January 2017
Supervisor: Sverre Hendseth, ITK

Norwegian University of Science and Technology
Department of Engineering Cybernetics

Title: Kybulf jr. The Walking Six-legged Bug Robot

Student: Edvin Andreas Holmseth

Problem description:

Design and build a functional hexapod. The hexapod should be able to walk in all directions and be controlled by a handheld controller. When turning, the body should not rotate, but rather change the side that acts as the "front end". The robot must be able to communicate with a computer for future possibilities of implementing an autonomous control system that receives orders from a computer.

Master of Science in Engineering Cybernetics

Submission Date: January 2017

Supervisor: Sverre Hendseth

Abstract

This paper describes how to build a small, lightweight hexapod (a six-legged bug-robot), using mostly off-the-shelf electronic hardware.

The paper will discuss the electronic components that are used and their role in the robot. It explains how the robot was designed, 3D-printed, and how to assemble the robot.

It will also take a closer look at how the software developed for the robot is built up, and how the different hardware components communicate with each other. Finally, we will explain how the robot controls the legs, both in a mathematical view and from a software perspective.

Sammendrag

Denne oppgaven beskriver hvordan man kan bygge en hexapod (en seksbent insektrobot), ved å bruke hardware som for det meste kan kjøpes på nett eller i butikker.

Oppgaven vil diskutere hvilke elektroniske komponenter som er brukt, og hvilken rolle de har i roboten. Den vil også forklare hvordan robotens design ble valgt, 3D-printet, og hvordan man setter sammen roboten.

Vi vil også se nærmere på hvordan softwaren som ble skrevet for roboten er bygget opp, og hvordan de ulike hardwarekomponentene kommuniserer med hverandre. Til slutt vil vi se hvordan roboten kontrollerer benene sine, både fra et matematisk synspunkt, og i et software-perspektiv.

Contents

Chapter 1
Introduction

1.1 Motivation

Both human controlled and autonomous robots have been around for many years. Most of them are propelled by wheels but during the last few years the interest in quadcopters, or "drones", has increased. A quadcopter can reach places wheel based drones can not, but there are also places that quadcopters have difficulties reaching, e.g. for rescue searches in collapsed buildings. In these places, a hexapod would do better than both flying and wheel based drones.

There are many robotics projects done at NTNU, resulting in robots in different shapes and sizes. The goal of this project is to create a robot that is small enough to bring around to do demonstrations while promoting the engineering studies at NTNU e.g. in Upper Secondary Schools. Another goal is not only to build the robot but to learn how to control a walking robot using source code only, with no help from other software, such as Matlab or similar mathematical programs.

1.2 Outline of the Thesis

In this paper we will see how the robot was developed. We will take a look at all the hardware, including the electrical hardware, and the design and production of the plastic body. A guide on how to assemble the robot will be presented. We will see how the legs are controlled, both in a mathematical view and through the software.

Chapter 2
Background

2.1 Previous Work

The hexapod is not a new invention. It has been made by many people before, both commercial and in private. There is also a master's thesis from NTNU, written many years ago, that made a hexapod. Unfortunately, that master's thesis was difficult to find for the use of reference since it was written before NTNU started using Daim, and the Institute has archived their old library. The old robots name was Kybulf. When Kybulf was made, the electronic components were a lot larger than they are today. The power source e.g. was so big that it had to be carried by a person behind the robot. The goal of this thesis is to make a smaller, lightweight version of Kybulf, called Kybulf jr. Since the thesis on Kybulf was not found, nothing is reused, other than the idea.

The work on the hexapod started in the spring of 2016 with the subject TTK4550. The main part of this subject was to find out whether it was possible to build a hexapod using only parts that are commercial, in addition to some 3D-printed parts. The conclusion of this project was that it was possible, and the preparations for building the robot started.

By the summer of 2016 the hexapod, Kybulf jr, consisted of one leg with two motors that to some degree was controlled a microcontroller. The same microcontroller that is used in the final version of Kybulf jr. The pre-project also led to knowledge on PWM (Pulse Width Modulation) Motors. In addition to running two PWM motors on the Discovery STM32F4 microcontroller, one USART communication line was set up so that the microcontroller could communicate with a computer. This was used to change the PWM signals for the PWM motors in real time.

The rest of this chapter will give a short brief in how USART communication and PWM motors work.

2.2 USART Communication

USART communication, or serial communication, is a communication protocol used between electronic components. It is a fairly simple way of communicating. A USART port consists of a transmission line (Tx) and a receiver line (Rx). These are always cross-wired so that the Tx port on one component is always connected to the Rx line in the other component, and vice versa. See figure 2.1.

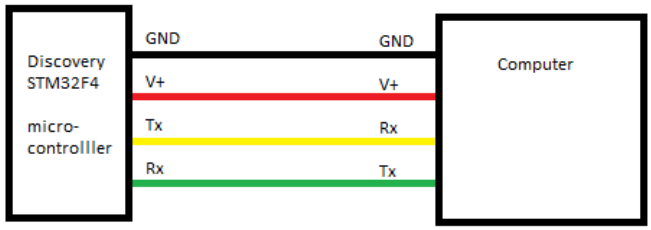

Figure 2.1: Wiring a USART Communication Port

For the communication to work, a fixed baud rate must be set on both sides of the communication line. The baud rate represents the speed of which the bits are sent. Typical baud rates are 9600, 19200, 38400 and 115200 [bits/sec]. In this project the bit rate 9600 bits/sec is used in all the electronic components.

The communication is always sent one bit at a time. If you try to send a message string, it is divided into individual bytes, consisting of 8 bits. Attached to each side are a start and a stop bit. On the receiving line, the device is waiting for the signal to drop from HIGH to LOW, which would be the start bit. When it drops, the device knows it will be followed by eight payload bits and a stop bit in the end, which is always HIGH. See figure 2.2. When all the bytes are received the message string is reassembled to a complete message.

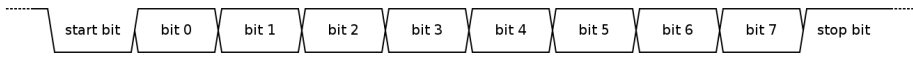

Figure 2.2: USART Message [17]

2.3 Pulse Width Modulation (PWM) Motors

Pulse width modulation motors have an integrated regulator that controls the motor. They use a pulse signal to control the output angle. The pulse signal is sent to the motors directly from the Discovery board in this project, while a separate power source is used for the motors. See figure 2.3.

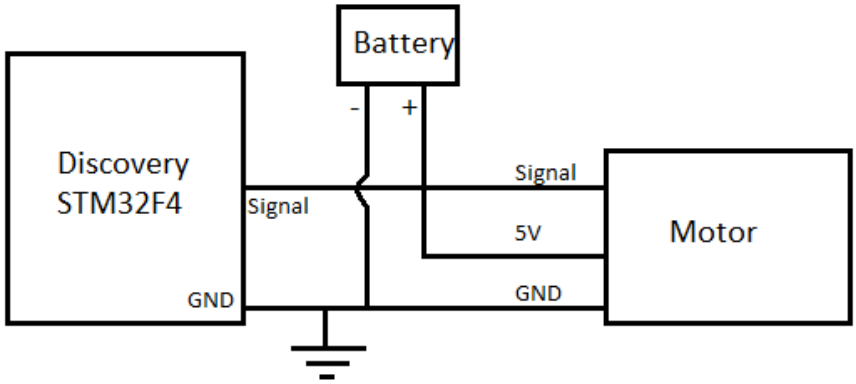

Figure 2.3: PWM Motor Wiring Scheme

The PWM pulse signal sent from the Discovery microcontroller is HIGH for a certain amount of time called the "duty cycle". Then it drops to LOW for the rest of the period. A normal period for PWM signals for motors of this size is 20ms. The length of the duty cycle is usually from 1ms to 2ms, where 1ms sets the rotor to 0°, while 2ms sets the rotor to 180°. Naturally, all duty cycles between the two endpoints can be used, and they correspond to the angles in between 0° and 180°. See figure 2.4.

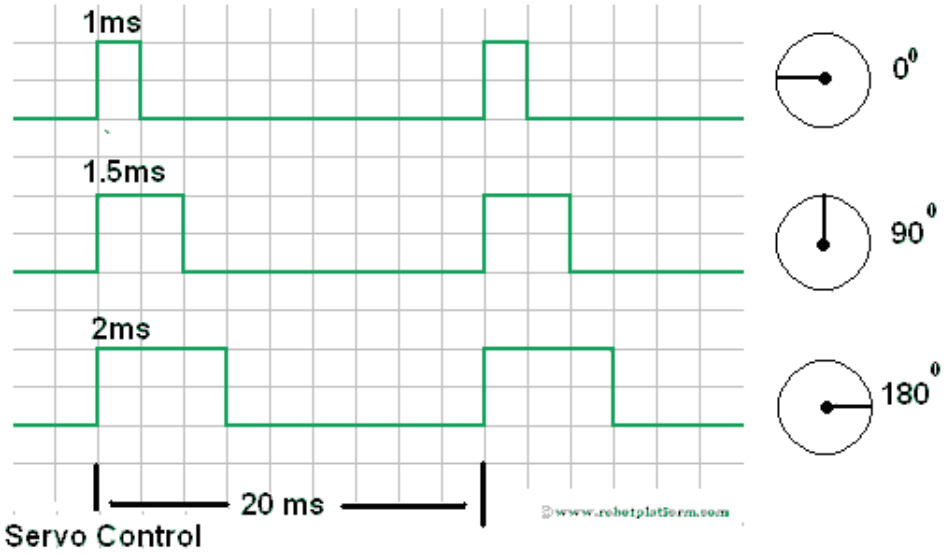

Figure 2.4: PWM Signals Explained [4]

2.4 Learning Outcome

This project has had many different aspects, many of which is fairly known to an Engineering Cybernetics student, while others might be new. One of the things I have learned during this project is using 3D-modeling software, which in this project was SolidWorks. I had never used any 3D-modeling software before, but online tutorials helped a great deal in getting started. Even though the parts in the robot are quite simple, assembling everything into a full model of the robot took a while, but was very neat to have a 3D-model of the robot, or parts of the robot, to create illustrations for this thesis. The 3D-modeling was also, of course, crucial to create the parts that needed to be 3D-printed, and to ensure that they would fit together before printing them.

Even though I was familiar with software development, it was a challenge to create such large amount of source code from scratch. I think that if I were to write it all again now, it would look a bit different, which is an experience I will bring into my career. Although I have programmed microcontrollers before, in several subjects at NTNU, I have never written much of the communication between electronic components, such as USART communication or PWM signal generation, since they are often included in a hardware abstraction layer on many microcontroller starting kits, such as Arduino.

Chapter 3
Background: Electronic Hardware

For this project, mostly commercial off-the-shelf hardware is used, so that it is easy to build a replica of the robot if needed. Except for the body and the legs, which is 3D-printed here at NTNU, everything else, like the microcontrollers, the power supply, the PWM motors and the radio remote controller is off-the-shelf products. A part that is not printed nor ordered is the power distributor, used to distribute power from the battery to the motors. It was made in the Engineering Cybernetics workshop at NTNU. It is a quite simple part, and it should not be difficult to create another if needed.

3.1 Discovery STM32F4 - Microcontroller

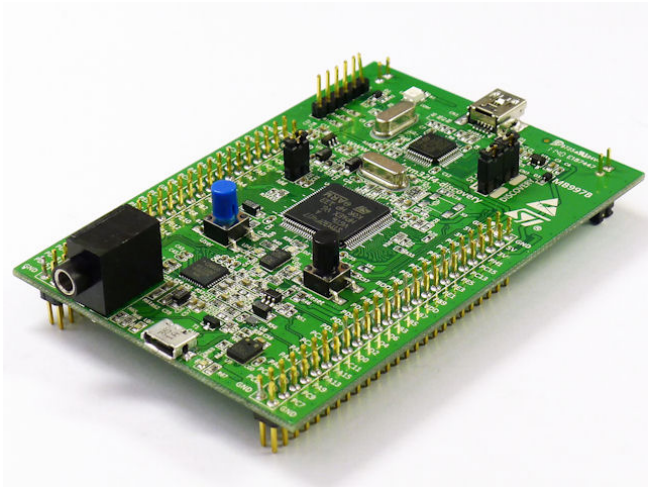

Figure 3.1: Discovery STM32F4 Microcontroller [2]

The main microcontroller used in this project is the Discovery STM32F4 microcontroller. There are plenty of microcontrollers that would be suited for this project, but this one was already available when the project started. The microcontroller has many IO-ports, which is needed to control the 18 PWM motors The IO-ports are also required for USART communication, which is used with the remote radio controller receiver, and communication with a computer, which is necessary during the phase of testing.

3.2 Motors

Figure 3.2: Tower Pro SG90 9g [3]

The motors used for the project is the Tower Pro SG90 9g micro servo motors. this motor is a PWM motor (Pulse Width Modulation motor). It was decided to use PWM motors, as they are reasonably cheap, and they have a built in regulator, which makes them easier to control directly from the microcontroller, which again simplifies the software and hardware.

For an introduction on how PWM motors operate, see chapter 2.3. Notice however that the TowerPro SG90 9g servo motors use different duty cycle limits than the ones used in the Background chapter. The correlation between the duty cycles and the output angles are not completely linear, but nearly perfect linear from -60° to +60°, with duty cycles 700μs to 1800μs respectively, and with the center, 0° at 1250μs.

3.3 Telemetry Kit

The robot can communicate with a computer via serial communication. It can send and receive message strings, which was very useful during the software development, both to see and to change different parameters while the software was running. There is not developed software for the computer to communicate with the robot, but a simple program, such as Termite [6] can read and send the text string messages in real time.

There are many solutions on how to connect IO-pins on a microcontroller to a USB port on a computer. A USB to serial adapter can be used, but this would require a wired connection between the computer and the robot. Instead, a Telemetry kit from 3DRRadio is used. See figure 3.3.

Figure 3.3: 3DR Radio Kit [7]

The 3DR Radio kit offers communication as if there was wired serial connection between the robot and the computer. A configuration software can be downloaded from ardupilot's website [8], and it is also included in the zip folder attached to this paper. Note that for the telemetry kit to work, the baud rate of the radio must be set to the same baud rate that is used in the USART communication module in the

software on the Discovery STM32F4. The transmission power (Tx Power) has also been increased to minimize the data loss as the distance between the sender and receiver is increased. All the settings can be seen in figure 3.4.

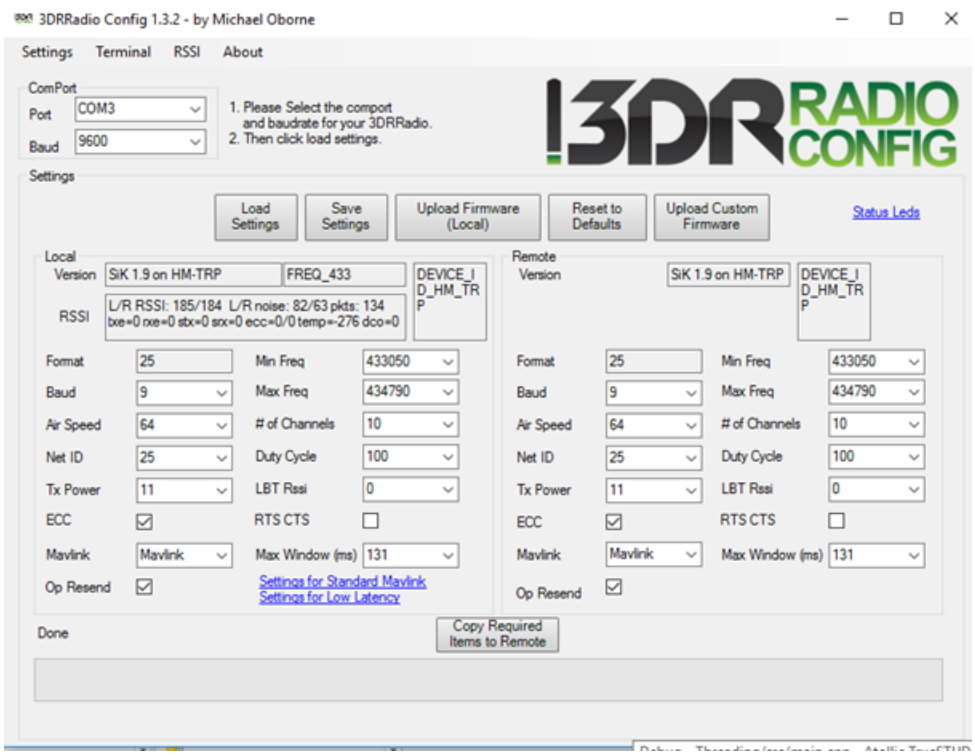

Figure 3.4: 3DR Radio Config

3.4 Radio Controller

Figure 3.5: Flysky FS-T4B, Radio Controller [5]

The robot is controlled with a handheld radio controller. The controller used in this project is the 2.4G 4CH Radio Control Transmitter for Flysky FS-T4B [5]. It comes with a receiver that translates the radio signals to PWM signals, which can be measured by a microcontroller. Both the controller and the receiver are pictured in figure 3.5.

The radio controller works pretty much the same way as a PWM motor, as they often work with PWM motors without any microcontrollers in between in RC cars and RC planes. The controller transmits a radio signal to the receiver, which then reads the signal, and transforms it into a PWM-signal. The controller has two joysticks, which each can be pushed in two axes. Each axis is read separately by the receiver, which translates it into four PWM signals, which is read by a microcontroller.

The range of the duty cycles in the radio controller is similar to a standard PWM signal. It has a period of 20ms, but the duty cycles seem to vary a little in the different channels. The center position of each joystick axis corresponds to duty cycles ranging from $1414\mu s$ to $1508\mu s$, and the endpoints vary a lot. This is taken

care of by the software. The raw values from the radio controller can be seen in Table 3.1. Also, note that the channels marked with "RUD.", "THR." and "ELE." are flipped on the radio controller, so that the up or the right position on each joystick axis corresponds to the highest duty cycle on all the channels.

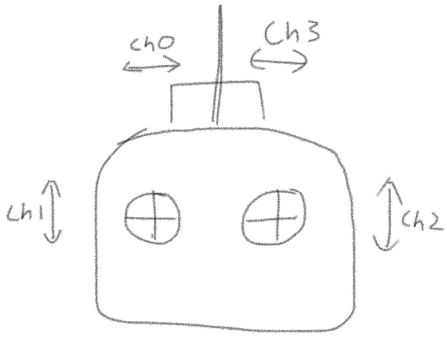

Figure 3.6: Radio Controller Joystick Channels

Channel	Duty cycle mid[μs]	Duty cycle min[μs]	Duty cycle max[μs]
0	1480	1050(left)	1950(right)
1	1414	1120(down)	1770(up)
2	1480	1080(down)	1900(up)
3	1508	1070(left)	1960(right)

Table 3.1: Output values for radio controller PWM-signals

3.5 Arduino Nano - Microcontroller

Because 18 of the PWM ports was used to control the motors on the robot, there were not sufficient PWM ports to read all the PWM inputs from the radio controller. A separate microcontroller was needed to measure the PWM inputs. An Arduino was chosen because it is lightweight and cheap. The Arduino measures the duty cycle of all four PWM signals from the radio controller receiver and sends the values to the Discovery STM32F4 via serial communication.

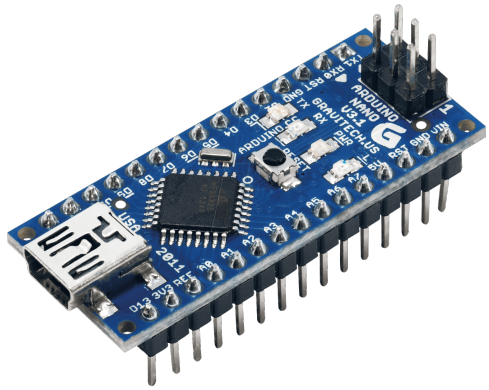

Figure 3.7: Arduino Nano [9]

3.6 Power Sources

As the motors require 4.8V and the Discovery STM32F4 requires 5.0V, separate power sources are needed. To power the Discovery board a simple Clas Ohlson power bank [11], as seen in figure 3.8, is used. However, while the servo motors can run on 5V, this power bank is not sufficient to run 18 servo motors at once. Therefore a separate battery for the motors is required. A Bronto Rx Pack [10], as seen in figure 3.9, is used to power the motors in this project.

The power bank is connected to the Discovery microcontroller via a USB to mini-USB cable, while the Bronto Rx Pack is connected to the motors via the power distributor made in the Workshop for Engineering Cybernetics at NTNU.

Figure 3.8: Clas Ohlson Power Bank [11]

Figure 3.9: Bronto Rx Pack [10]

3.7 Power Distributor

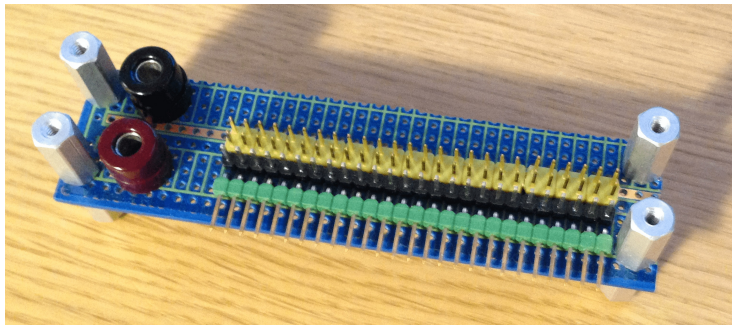

Figure 3.10: Power Distributor

The power distributor is made at the Workshop for Engineering Cybernetics at NTNU. It is a circuit with four rows where on the two first rows, all the nodes on that row are connected to each other. One row for GND and one row for V+. The last two rows are connected to each other to carry on the individual PWM signals from the Discovery STM32F4. The PWM signal pins are included in the power distributor because all the PWM motors come with a "PWM contact" as in figure 3.11, with the three inputs: "GND", "V+", and "Signal", in that order. This way, the motors can be plugged directly into the power distributor, along with each PWM signal from the Discovery STM32F4 connected next to it. When the battery is connected to the power distributor, and all the PWM signals from the Discovery

microcontroller is connected, the motors get both power and the control signal from the power distributor.

Figure 3.11: PWM Servo Motor Contact

A simple circuit drawing of the power distributor can be seen in figure 3.12. All the squares represent pins. The gray pins are connected to one another (GND). The same goes with the red(V+). The numbered squares are connected to the squares with the same number. 1 is connected to 1, and 2 is connected to 2, and so on.

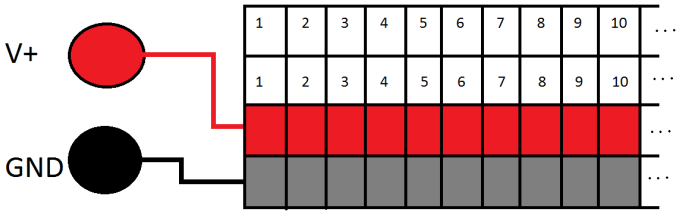

Figure 3.12: Power Distributor Circuit Drawing

Chapter 4
The Robot's Design

4.1 Body

One of the goals for the robot was that it should be able to walk in all directions without rotating the body. Therefore the robot has no natural front or back end, and thus, it has no head. The body is completely round with six legs with a 60° angle between each leg.

The body consists of layers. There is a main layer containing the motherboard (the Discovery STM32F4), the leg mountings, the telemety kit, and the radio control receiver. On top of this layer, there is another layer containing the batteries and the power distributor. The reason for this is so that all the heavy components are equally distributed around the center of the body so that the weight is equally distributed on the legs. This also leaves room for further development. If desirable, one can add a new layer with sensors etc, for automatic control of the robot. The concept sketch of the body is seen in figure 4.1, where the Discovery board is located in the first layer, and the battery is located in the second layer. The size of the body was chosen by the size of the motherboard, and the size of the motors.

4.2 Legs

It was important for the legs to have three degrees of freedom so that the endpoint of each leg can be moved anywhere in space. For this, three motors in each leg are required. The design that was chosen was based on an ant's leg. The inner motor is moving sideways, and the middle motor is moving up and down. The first two motors make the "hip" joint of the leg. The outer motor, placed on the "knee" is also moving up and down. See figure 4.2. The length of the inner leg, that is, between the 2^{nd} and 3^{rd} motor is 10cm long. The link outside the 3^{rd} motor is 15 cm.

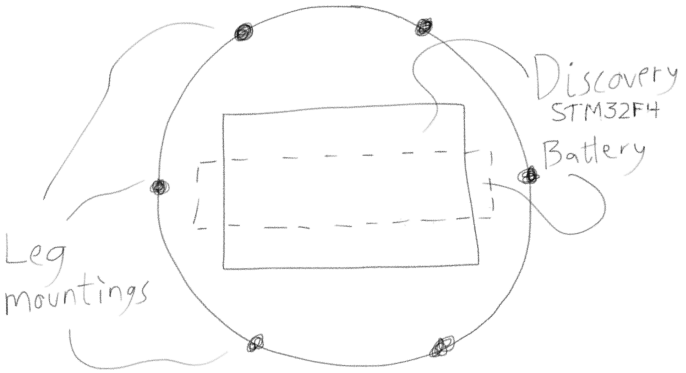

Figure 4.1: Conceptual Sketch of the Body

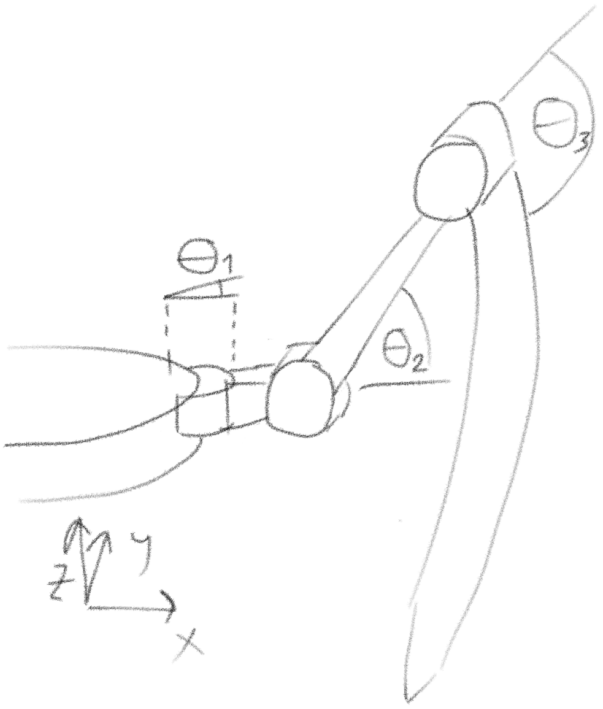

Figure 4.2: Conceptual Sketch of the Leg

Chapter 5
3D-Modeling and 3D-Printing

During the phase of development, a great way of visualizing the hexapod was to use a 3D modeling software to design the robot. SolidWorks is a 3D-modeling software used on NTNU, and it was used in this project both to visualize the robot and to design the parts of the body that were later 3D-printed.

5.1 Motors

The Tower Pro motors have a simple shape, and a 3D model of them was found online at GrabCad [15]. A simple motor casing was designed to fit around the motors so that the motors would attach to the body. The motor casing can be seen in figure 5.1. The motor casings were designed a little different for each of the motors on the leg. The casings for the two inner motors were attached together in one piece and designed to fit into the first layer of the body, while the outer motor casing was designed to fit onto the outer leg part, but they all have the same basic layout, as seen in figure 5.1. The specifications of each motor housing can be seen in appendix B.

5.2 Body

As mentioned before the body's design is based on a layer module. The main layer is the layer where the inner joints and the motherboard are placed. The layers are based on a simple plate that can be modified to house different hardware. In figure 5.2 we see the two plates that make the first layer. The leftmost is the bottom plate and is equipped with a card holder and some extra holes to fit the motor sockets. The rightmost base has specially formed holes to fit the rotor part of the PWM motors. In figure 5.3 we can see what the first layer looks like, including the Discovery STM32F4, the motor casings, and the motors. Notice how the rotors on the motors in figure 5.3 fit in the holes on the right side in figure 5.2.

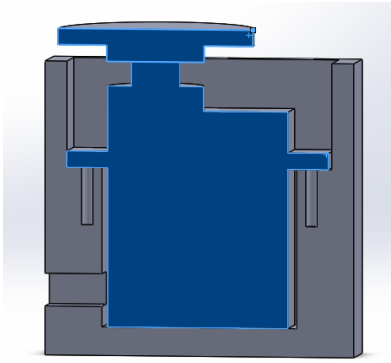

Figure 5.1: Motor with Casing (Cutaway view)

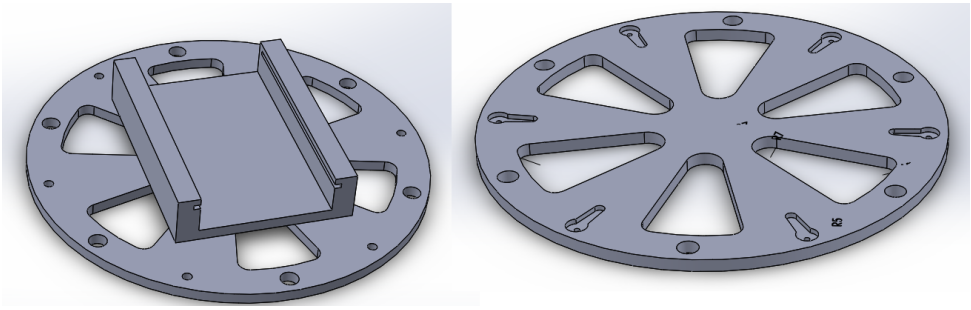

Figure 5.2: Base Layer

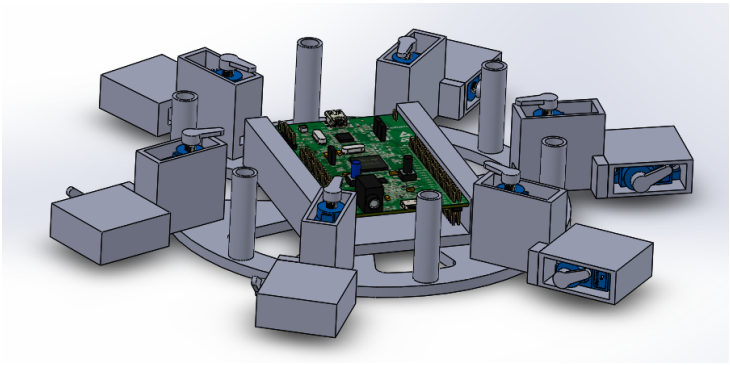

Figure 5.3: First layer of body

5.3 Legs

As mentioned, the legs each have three joints, which means three motors. We call them motor 1, 2 and 3 for simplicity, where motor 1 is the motor closest to the body, motor 2 is the middle motor, and motor 3 is the outer motor. Motor 1 is connected to the body, and motor 2 is connected directly to motor one, with no link in between, other than the motor casings themselves. Motor 2 is connected to a link which makes the "middle leg" part, which then is attached to motor 3, which acts as the "knee". Motor 3 is connected to a final link, which is the "outer leg", and leads to the endpoint of the leg. See figure 5.4.

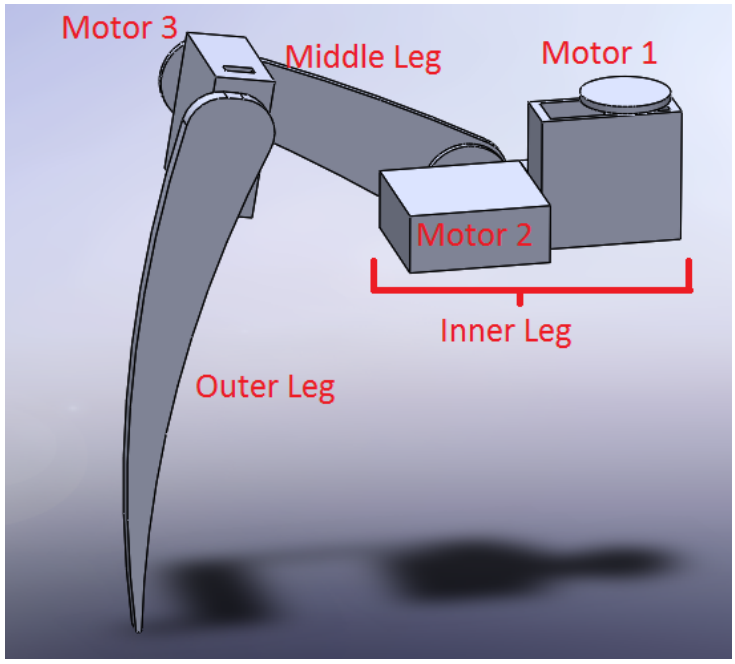

Figure 5.4: Robot Leg

5.4 Full Model

In figure 5.5 we can see the full 3D model of the robot. The parts that are not included are the wiring and the batteries, which were not yet ordered when the parts were set to printing.

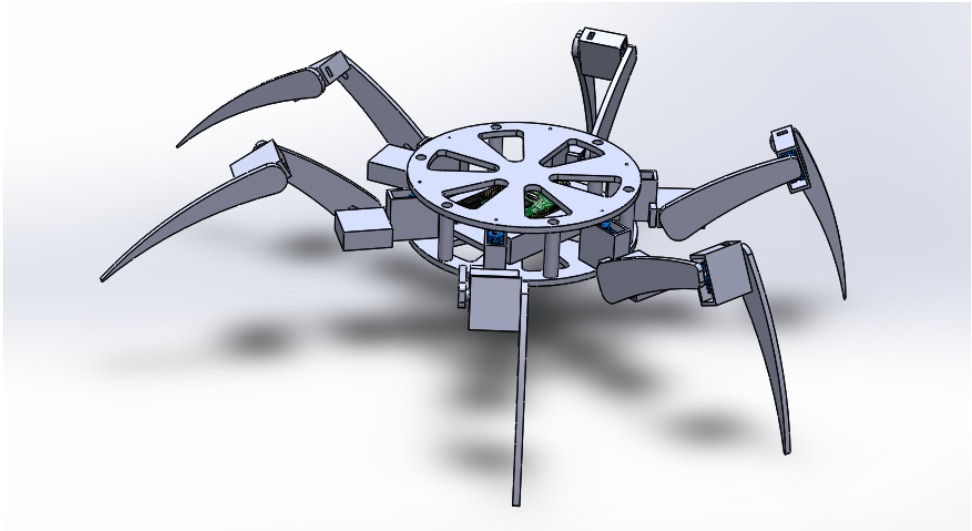

Figure 5.5: Full Model

5.5 3D-Printed parts

The 3D-Printing was done at the workshop at the Institute of Engineering Cybernetics at NTNU. The process took about 30 hours. The parts came out pretty much like expected, and they fitted each other like they were supposed to.

5.5.1 Motor Sockets

During the pre-project last semester, some motor sockets was printed, and they fitted the motors fine. This time, however, some of the motors was a bit difficult to get into the sockets, due to minor varieties in each printed socket. All of them worked in the end, but if someone is to print new motor sockets, a little larger space for the motors is recommended.

5.5.2 Body

The card holder for the Discovery STM32F4 printed directly on the bottom piece works very well, as the Discovery board fits perfectly in place. Only the friction is enough to hold the card in place. The cylinder spacing pieces seen in figure 5.3 was not printed but made out of a solid plastic rod. In retrospect these pieces could have been made in the same part, that is, attached to the bottom body piece. By doing that, they can be hollowed out, so they weigh less, and it would be less work in the workshop. If someone is to print the bottom layer once more, it is recommended to

add the cylinder spacers to this piece. It would also be beneficial to see if there is a way to hollow out parts of the card holder to minimize the use of plastic, and thus reduce the weight, since the card holder is not a part that has to withstand much force.

5.5.3 Legs

The leg pieces also work very well. The indents made for attaching the leg parts to the rotor of a motor, or to the motor socket works great. The only modification that needed to be done after printing them was to drill holes in the indents made for the rotors, to fasten them to the motor. These holes could have been included in the 3D-model, but one would have to drill bigger holes in the rotor parts anyway, to make the holes big enough to fit a screw into it. The outer leg parts could have been attached directly to the outer motor sockets, instead of making indents and extruders. This could, however, lead to a longer printing process, as the pieces would take up larger space in the printer, thus having to set several different prints, instead of stacking the neatly into one print.

The legs seem to hold the weight of the robot, but they tend to bend a little, especially around the middle motor. The solution used to hold the inner motors, with connections on both sides of the rotational axis, seems to work better than on the two other motors. If someone is to design new motor casings, a similar solution to the one used to attach the inner motor casings is advised for the two others. This does, however, increase the weight of the robot. The current design works fine, but it was made without knowing the strength of the plastic in the rotors of the motors, and it is not optimal.

Figure 5.6 shows some photos of some of the 3D-printed parts.

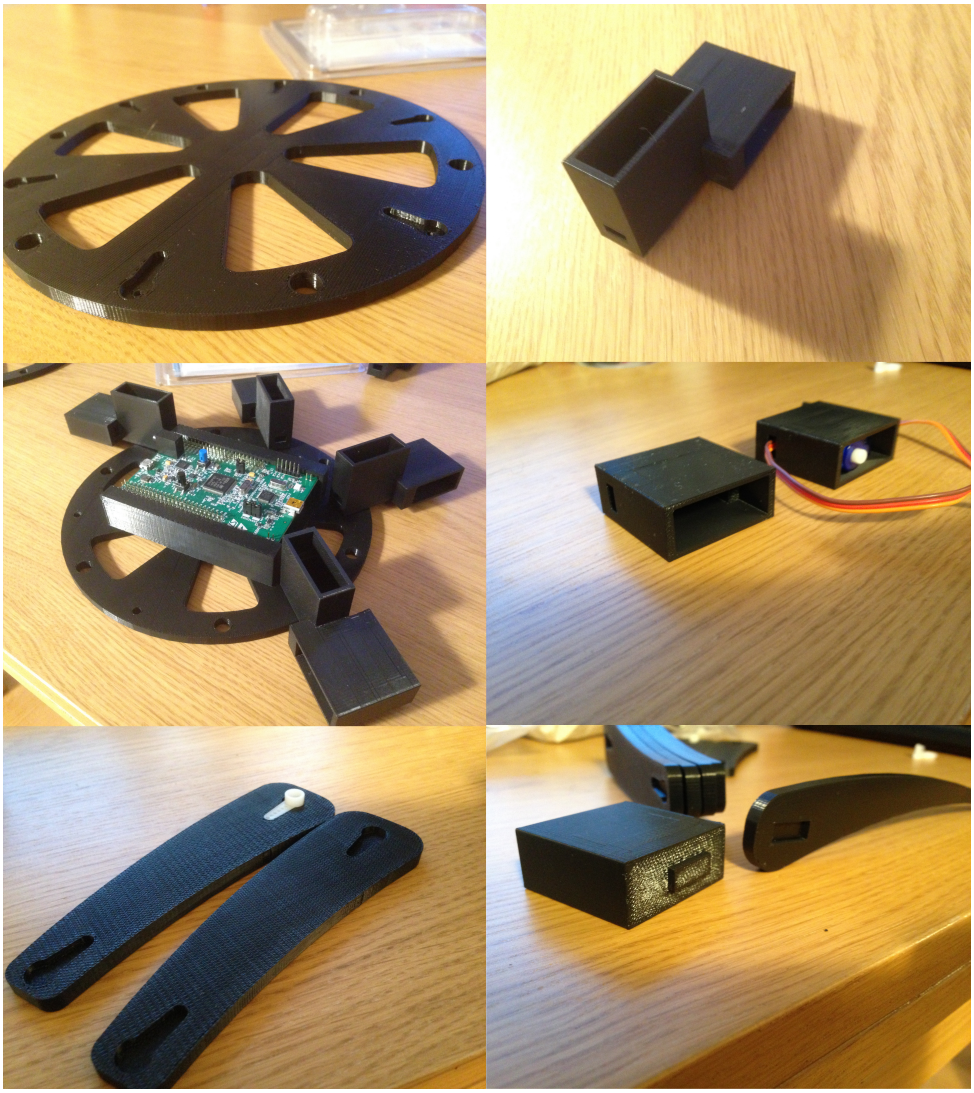

Figure 5.6: 3D-Printed Parts

Chapter 6
Assembling the Robot

6.1 Checking the Motors

Before assembling the robot, make sure that all the motors are working, and that they are equally adjusted from the factory. This can be done by using the Discovery STM32F4. However, this requires some modification of the software. In the file MotorControlModule.cpp, comment out the lines: "this->UpdatePwmInput();" and "this->RunLegController();", and uncomment "this->UpdatePwmFromInput();". All the PWM duty cycles are now set to 1250, which is the middle position for the TowerPro motors. The duty cycles can also be modified via USART communication commands in Termite in real time. See appendix C.6.2 to see where in the source code to apply the changes to test the motors. To change the PWM duty cycles, use the commands described in appendix A. Use this to check that a duty cycle of 1250 μs corresponds to the middle position of the motor, and that they all move the same amount with different duty cycles. When all the motors are checked, fit them into their motor casings.

6.2 Assembling the Legs

The rotor part on the motors, circled in red in figure 6.1, are disconnected when the motors are delivered. It is important to connect them correctly so that the angles actually are the same as the software assumes, as there is no feedback from the motors to the Discovery STM32F4. Before connecting them, we need to drill some holes for the screws. Place each of the rotor parts into the slots where they are supposed to be on the robot, like in figure 6.2. Then drill a hole through both the rotor part and the part where the motor is intened (e.g. the leg or the body), so that a bolt can fit in. Two holes are needed for each rotor part. The final result will look like the picture in figure 6.3.

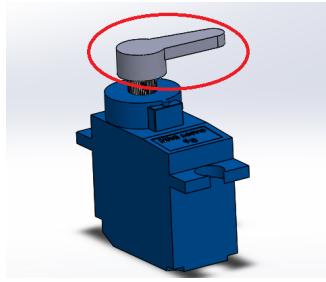

Figure 6.1: Rotor Part

Figure 6.2: Modifying the Rotor Parts

Figure 6.3: Attached Motor

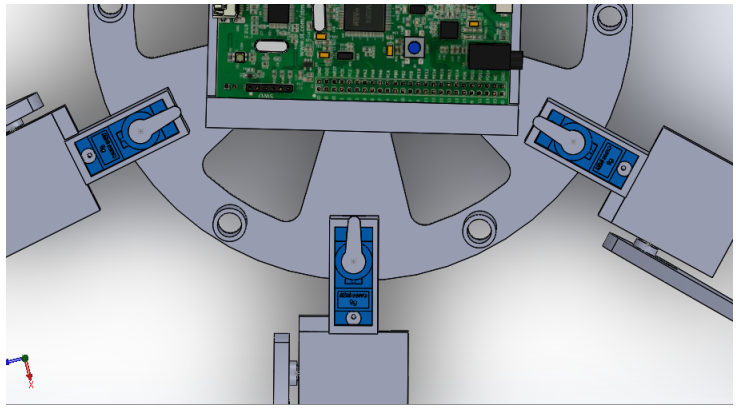

Figure 6.4: Positioning of rotor on inner motor

The rotor parts on the inner motors, that is, the ones closest to the body on each leg, should point straight inwards when the PWM signal (duty cycle) is 1250μs. See figure 6.4. Thus the legs can move the same amount in both ways. If a power supply or a PWM signal is not available, one can find the middle position (1250 μs) by twisting the rotor to the end in both ways to find the middle, but this is not as accurate as using the Discovery STM32F4 to generate a duty cycle of 1250μs. What pinouts to use for the duty cycle output can be seen in table 6.1. Choose a pinout that corresponds to one of the inner motors, such as motor 1,1.

The rotor part on the middle motors of each leg should be connected so that the rotor part is pointing straight out from the body when the motor receives a duty cycle of 1250μs. See figure 6.5 for illustration.

The rotors on last motors, the outer motors of the leg, should be pointing at a right angle to the left when the motor is seen from "above" when it receives a duty cycle of 1250μs. See figure 6.6.

Now that all the rotors are attached correctly, screw them on to the leg pieces. The outer leg piece is glued onto the outer motor casing. The result should look something like the picture in figure 6.7.

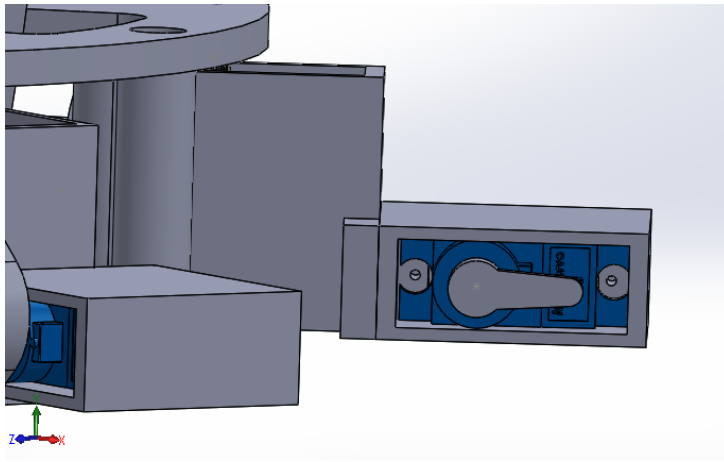

Figure 6.5: Positioning of rotor on middle motor

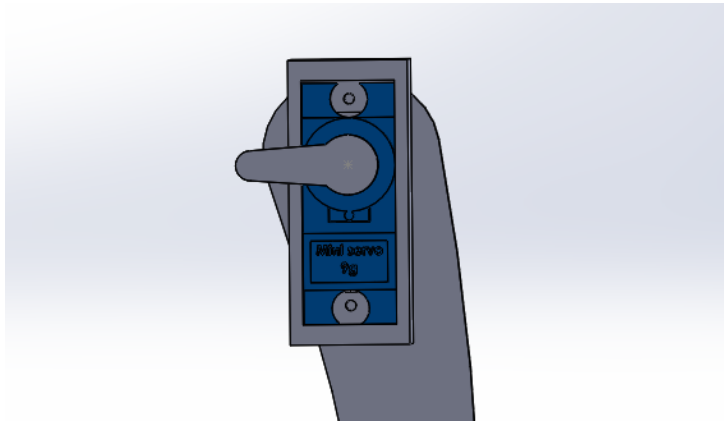

Figure 6.6: Positioning of rotor on outer motor

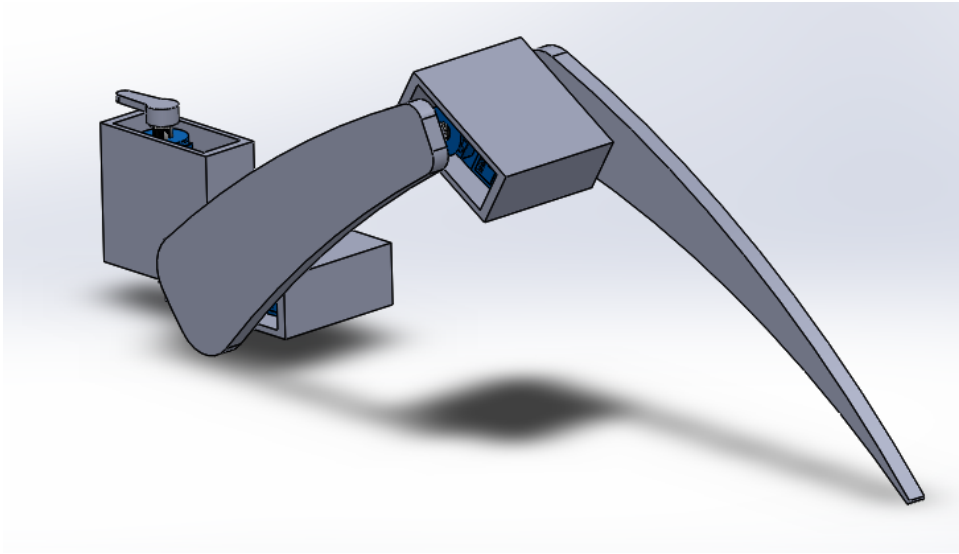

Figure 6.7: Assembled Leg

6.3 Assembling the Body

The first step in assembling the body is to slide the Discovery STM32F4 into the card holder. The friction holds the card in place, so there is no need to fasten it further. Note that the card holder blocks the view for the names of the two outer rows of the pinouts, as seen in 6.8. To see what name corresponds to what pinout, use figure 6.9. These pinouts can also be found online.

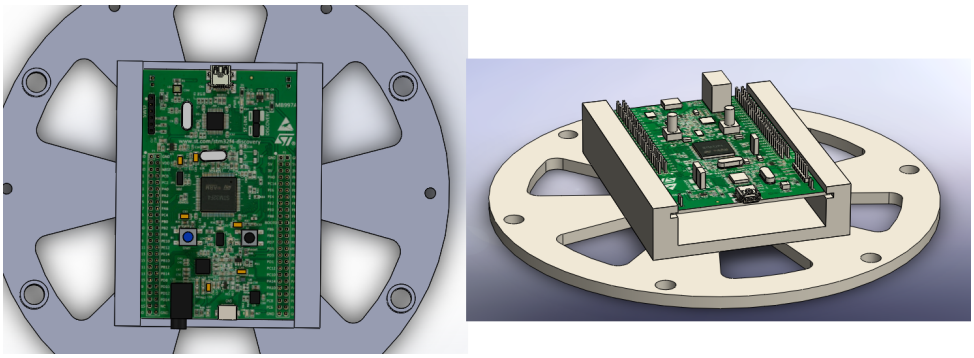

Figure 6.8: Discovery STM32F4 in card holder

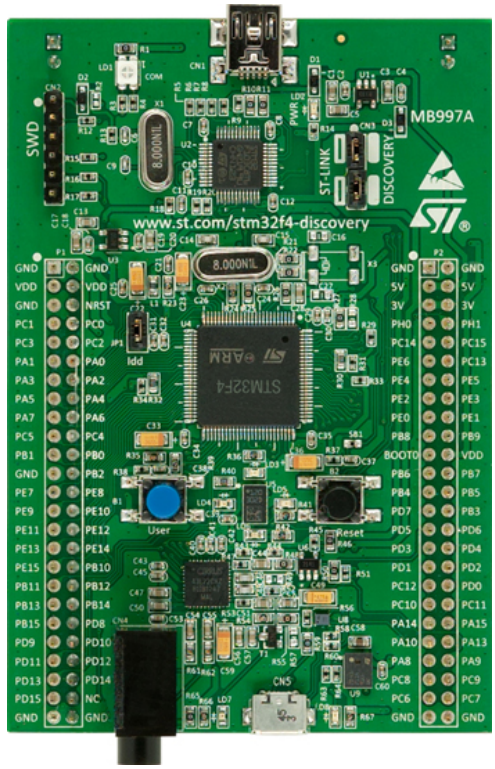

Figure 6.9: Pinout Overview Discovery STM32F4 [14]

Continue by connecting the spacing cylinders. These were not printed but made from a plastic rod in the workshop. It is recommended to print these directly on the lower base. Whether they are printed or attached, the robot should now look like the illustration in figure 6.10. Note that only four spacing cylinders are used, as one of them would block for the mini-USB cable that is connected to the power bank. The last one is excluded due to weight symmetry.

The next step is to attach the legs to the body. Start with the upper body part, the part without the card holder, and lay it upside down. Then screw on each of the legs in the rotor-shaped holes. See figure 6.11.

6.4 Wiring

Before attaching the lower half, with the Discovery board, to the upper half, with the legs, all the wiring needs to be connected, because it is very hard to reach the pinouts on the discovery once the body is assembled. This is a bit tricky because

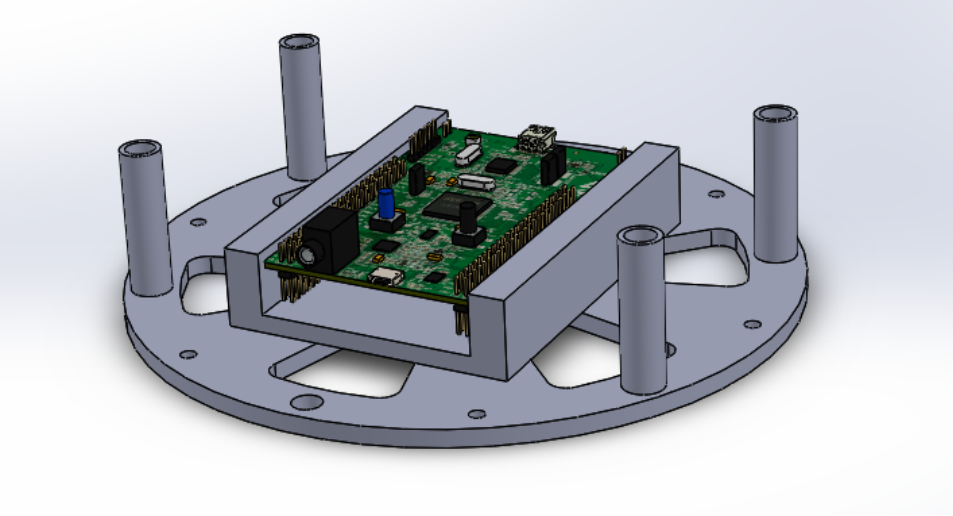

Figure 6.10: Assembling the Spacing Cylinders

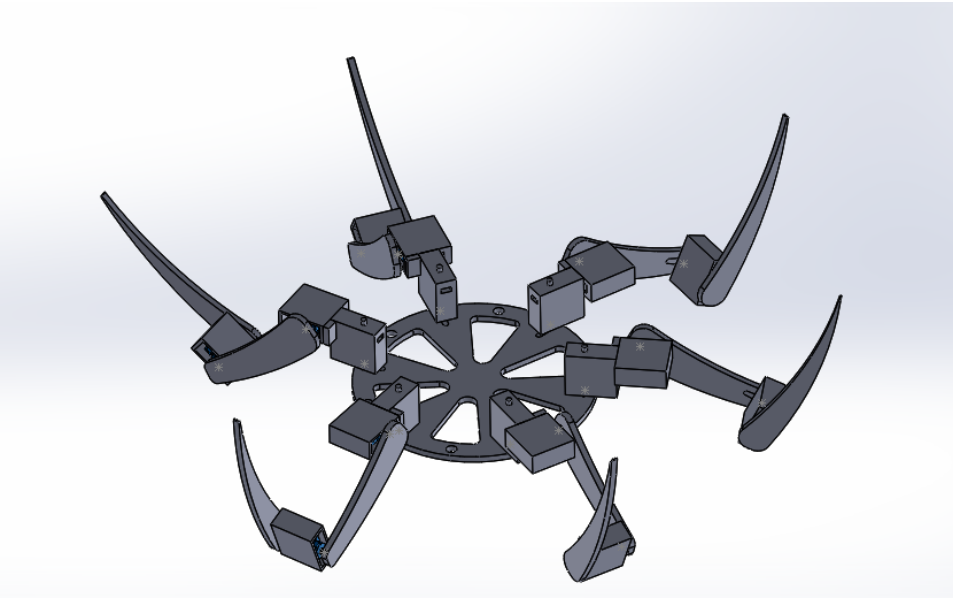

Figure 6.11: Assembling the Legs to the Body

the wires have to go through one of the holes in the upper part of the body. It is easy to loose track of which wires that goes where. It is recommended to plug all the PWM wires directly into the power distributor in the order found in table 6.1, and then slide it through the hole. Remember to connect the ground row on the power distributor to GND on the Discovery STM32F4.

The Arduino should also be connected according to the wiring scheme in table 6.1. The Arduino and the radio controller receiver can be placed in the same layer as the Discovery STM32F4.

On the 3DR Radio Telemetry Kit, the pinouts are shown in figure 6.12. Note that the Rx in on the telemetry kit is going in the Tx on the discovery and the other way around.

Figure 6.12: Pinout 3DR Radio Telemetry Kit

6.5 Final Assembly

Once the wiring is complete, attach the upper body to the lower body by screwing the spacing cylinders in place. Make sure all the inner motor casings, attached to the body are inserted to the holes in the lower body part. Connect the PWM motors to the power distributor. The motors have to be connected in the same order as the

PWM inputs on the power distributor. On each leg, motor number 1 corresponds to the inner motor, motor number 2 the middle motor and motor number 3 the outer motor. It is not important which leg that is chosen as number 1, but leg number increments counter clock wise. When The motors are connected, the robot is ready to walk. Connect the power bank to the Discovery, and then the battery to the power distributor. Use the left joystick to lower the legs, and the right joystick to start walking.

USART			
3DR transmitter	(on Discovery)		
Ground	GND		
V+	VDD		
Tx	PD5		
Rx	PD6		
Arduino	(on Discovery)		
Ground	GND		
VIN	VDD		
Rx	PA9		
Tx	PA10		
Arduino	(on Radio Remote Receiver)		
GND	-		
5V	+		
D2	S(ch4)		
D3	S(ch3)		
D4	S(ch2)		
D5	S(ch1)		
PWM Motors			
Leg	Motor	Pinout (Discovery)	Timer –> Channel
1	1,1	PB6	TIM4 –> CCR1
1	1,2	PB7	TIM4 –> CCR2
1	1,3	PB8	TIM4 –> CCR3
2	2,1	PB9	TIM4 –> CCR4
2	2,2	PA6	TIM3 –> CCR1
2	2,3	PA7	TIM3 –> CCR2
3	3,1	PB0	TIM3 –> CCR3
3	3,2	PB1	TIM3 –> CCR4
3	3,3	PA5	TIM2 –> CCR1
4	4,1	PB3	TIM2 –> CCR2
4	4,2	PB10	TIM2 –> CCR3
4	4,3	PB11	TIM2 –> CCR4
5	5,1	PA0	TIM5 –> CCR1
5	5,2	PA1	TIM5 –> CCR2
5	5,3	PA2	TIM5 –> CCR3
6	6,1	PA3	TIM5 –> CCR4
6	6,2	PE5	TIM9 –> CCR1
6	6,3	PE6	TIM9 –> CCR2

Table 6.1: Wiring Scheme

Chapter 7
Leg Control Mathematics

All the programming done in this project is written in C++. The program is then uploaded to the Discovery STM32F4, which generates the PWM input for each motor. Chapter 8 will take a closer look at how the walking algorithm works but first it is important to know how each leg is controlled.

The method used to control the robot in this project is to have a reference point of where one want the endpoint of the leg to be, and then calculate the angles required for each joint/motor to make sure the endpoint is where it is supposed to be. In an industrial robot, one could e.g. find the kinematic matrix for the joints, and then the combined kinematic matrix for the whole arm, and reverse it to find the angles in each joint. In this case, the legs only have three degrees of freedom, and hereby only three motors. To reduce the computational power, it is easier to use basic trigonometry to calculate the motor angles of each leg.

7.1 Calculating the Legs Motor Angles

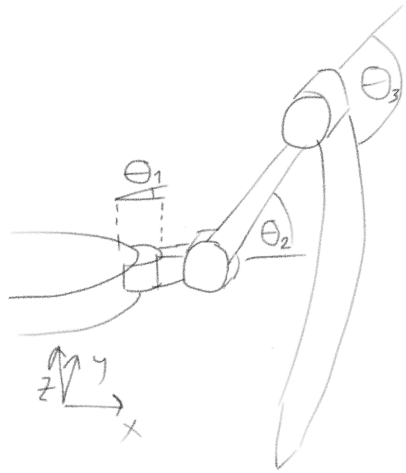

Figure 7.1: Angle Calculations

Figure 7.1 shows the different angles of the leg. To calculate the angles, the endpoint is given in coordinated in a body-fixed frame. The point where the leg is attached to the body is known, as well as the angle the leg is attached. First, the innermost angle is calculated, which is θ_1 in figure 7.1.

7.1.1 Calculation of Angle θ_1

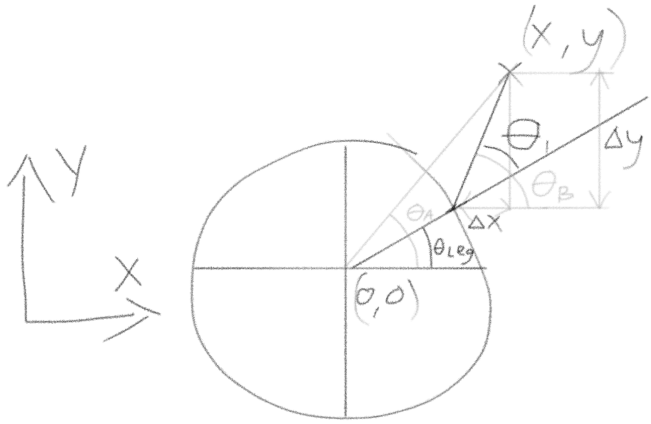

Figure 7.2: Calculation of θ_1

Figure 7.2 shows the relevant angles and distances needed to calculate θ_1. The point (x,y) is the point where we want the end of the leg to be, and it is given in a body-fixed coordinate system. Δx and Δy are then calculated, which is the distance from the base of the leg to the end point in the x and y-direction respectively. The C++ function atan2(double y, double x) is used to calculate the angle θ_B, which is the angle from the base of the leg to the end point given in the fixed body reference frame. By subtracting the angle from the center of the body to the leg, θ_{Leg}, from θ_B, the angle θ_1 is obtained.

7.1.2 Calculation of Angles θ_2 and θ_3

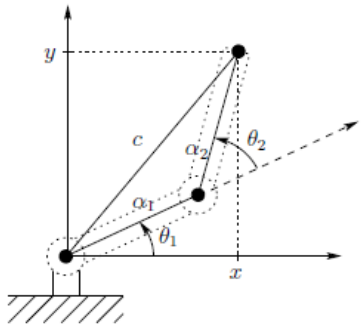

Figure 7.3: Calculation of θ_2 and θ_3 [12]

Figure 7.3 is from "Robot Modeling and Control - First Edition"[12], and is a good illustration for our angles θ_2 and θ_3, except that we want the "knee" to bend down, instead of up. Note that the angle θ_2 from figure 7.1 corresponds to θ_1 in figure 7.3, as well as θ_3 corresponds to θ_2. In the further derivation of the angles the names from figure 7.3 will be used.

To calculate these angles, the law of cosines (equation 7.1) is used.

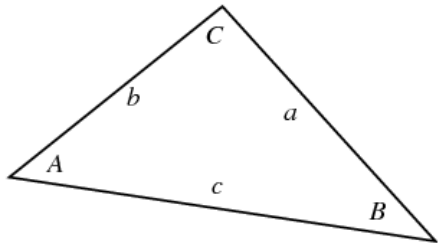

Figure 7.4: Law of Cosines [13]

$$cos(C) = \frac{a^2 + b^2 - c^2}{2ab} \tag{7.1}$$

By applying the law of cosines to figure 7.3, we find equation 7.2.

$$cos(\theta_2) = \frac{\alpha_1^2 + \alpha_2^2 - x^2 - y^2}{2\alpha_1\alpha_2} := D \tag{7.2}$$

Knowing that $cos(a) = cos(-a)$ the negative angle is always chosen, as it gives the angle of a knee bending down, which is desired. This gives $\theta_2 = -cos^{-1}(D)$.

To calculate the last angle, θ_1, the law of cosines is used once more, but now on the triangle created by the leg itself, as seen in figure 7.5. This gives equation 7.3.

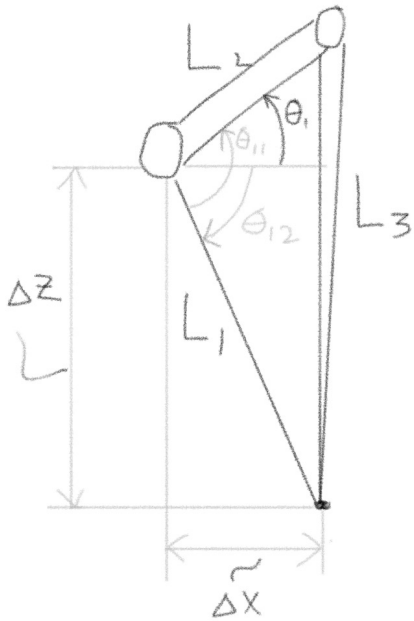

Figure 7.5: Calculation of θ_1

The angle of interest is θ_1, and the easiest way to obtain it is by calculating θ_{11} and θ_{12}. Notice that $\theta_1 = \theta_{11} + \theta_{12}$, θ_{12} being negative. L_1 and L_2 are the lengths of the leg links, and L_3 is given by the position of the base of the leg and the given

endpoint. The length $\Delta\tilde{x}$ is the combined length in the x and y-directions, projected into the leg plane.

$$cos(\theta_{11}) = \frac{L_1^2 + L_2^2 - L_3^2}{2L_1L_2} := E \tag{7.3}$$

$$\theta_{11} = cos^{-1}(E) \tag{7.4}$$

$$\theta_{12} = tan^{-1}(\frac{\Delta_y}{\Delta_x}) \tag{7.5}$$

$$\theta_1 = \theta_{11} + \theta 12 \tag{7.6}$$

Now θ_1, θ_2 and θ_3 can be fed into the motors, and the end point of the leg will end up at the given coordinates.

Chapter 8
Walking Algorithm

Now that the legs can be controlled, an algorithm to move the legs is required. The algorithm used in this project is quite simple. Each leg is given a stationary point in the body-fixed reference frame, equally distributed around the body, as seen in figure 8.1. The walking movement of the robot is controlled by the radio hand controller. Thus, the robot must be able to vary its speed and direction.

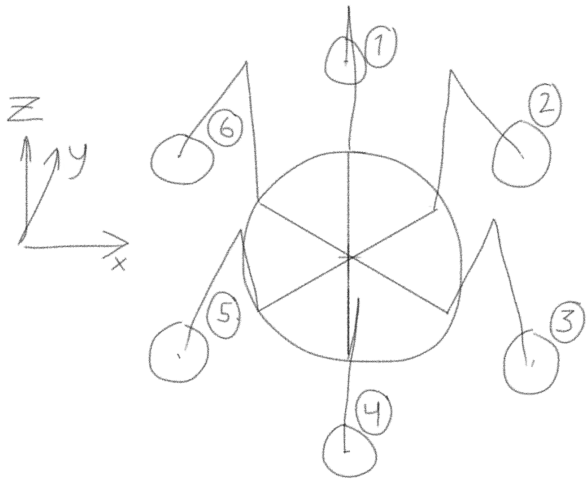

Figure 8.1: Stationary Points for the Legs

8.1 Walking Pattern

All the legs operate in their respective zones around their stationary points. The walking algorithm consists of moving the legs in a pattern in a separate coordinate

frame, with the origin located at the stationary point.

Each leg follows a triangle pattern as seen in figure 8.2. In this figure, the robot is walking to the right (in the y-direction). It starts by having the endpoint of the leg placed at the origin, point 0. Then the leg follows the trace along to point 1 (-L,0), where the length of L is determined by the software. When it reaches point 1, it will continue towards point 2 at (0,K), where K is determined similarly to L, and so on. When the leg reaches point 0, it starts over again. All the legs' endpoints pattern are parallel in the three-dimensional space, naturally, so that the robot is actually walking, in the direction of the Y-axis in this case.

The robot moves by always lifting three legs, like an ant. By giving a number to each leg, like in figure 8.1, the legs are divided into two groups. Leg 1,3 and 5 in one group and leg 2,4 and 6 in the other. One of the groups of legs is always touching the ground. To ensure that the legs that were in the air are touching the ground before the next leg is lifted, each step period is divided into four intervals. The endpoint of each interval is marked in figure 8.2. The two groups of legs follow the same pattern, but the groups are shifted two points compared to each other. This means that when one group is at point 1 and is about to lift itself off the ground, the other group is at point 3, and will take over the ground contact.

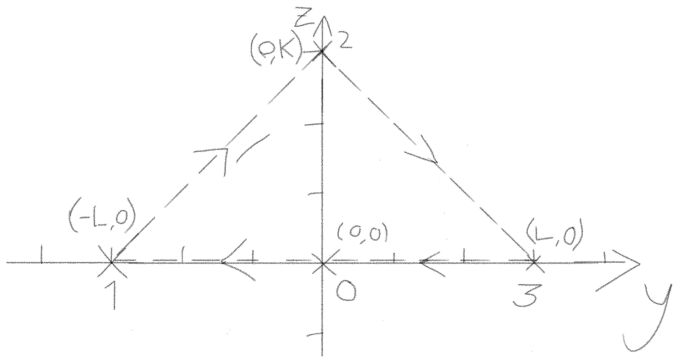

Figure 8.2: Walking Pattern

8.2 Changing Speed

When the robot does not receive a command to walk from the radio controller. All the legs are standing stationary on the ground in their respective stationary points.

As soon as the robot gets a signal to start walking, it starts to move the legs to the points given in figure 8.2. While there are different ways of adjusting the speed of the robot, such as slowing down the movement speed of each step, the method used in this project was to decrease the distances in each step. This is because when the two groups of legs are always on the opposite end of the triangle in figure 8.2, at least one of the groups of legs will always make a jump to get to the right position. When the speed is zero, the distance between the points in figure 8.2 is also zero. As the speed increases, the distance between the points increases proportionally.

8.3 Changing Direction

As mentioned earlier in this report, when the robot is turning, it does not rotate the body. Instead, it simply starts to walk in the desired direction, as if that was the new front end on the body. This is done by rotating the coordinates in the walking pattern in figure 8.2 in the three-dimensional space. See figure 8.3. First, the endpoint of each leg is calculated before it gets rotated around the z-axis in a coordinate system with the origin at the stationary point of that leg, using the standard rotation matrix for rotating around the z-axis. See equation 8.1

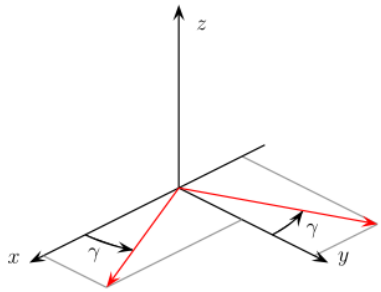

Figure 8.3: Rotation of Coordinates [16]

$$\begin{bmatrix} X_{\gamma,z} \\ Y_{\gamma,z} \\ Z_{\gamma,z} \end{bmatrix} = \begin{bmatrix} cos(\gamma) & -sin(\gamma) & 0 \\ -sin(\gamma) & cos(\gamma) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} \tag{8.1}$$

8.4 Summary

When implemented the leg movement of the robot is functional. The right joystick of the radio hand controller is used to steer the robot. As a startup safety, another fea-

ture was added to the software controlling the legs. When turning on the power on the robot, the legs are starting above the ground, so that the robot is laying on its "stomach". This means that the stationary points for the legs are placed a little higher than the bottom of the body on start up. The left joystick is then used to lower the legs. By pushing the left joystick down, the stationary points are lowered below the body, making the robot stand up, and ready to walk. In the zip folder attached to this paper, you will find a video showing how the robot is starting up, as well as how it responds to the different joystick inputs. The video can also be found on Youtube, using the following link: https://www.youtube.com/watch?v=KO0w32_NCcQ&feature=youtu.be

Chapter 9
Software Development

The Discovery STM32F4 is a microcontroller that uses the C++ programming language. Thus all of the software development in this project is written in C++. The microcontroller has some limitations, though. The Discovery STM32F4 has no possibility for threading, which we will take a closer look at in section 9.4.

9.1 Integrated Development Environment (IDE)

On the ST Microelectronics' (the developers of the Discovery STM32F4 microcontroller) website, some possible IDE's was listed. Among them was IAR, Atollic TrueStudio, Altium, Keil, and Cocoox. Some research showed Atollic, IAR, and Cocoox as the best options. IAR has a very limited free edition, where one could choose between a 30-day free trial, or a size limit of 30kb source code, which is very limiting if you intend to use a HAL (Hardware Abstraction Layer). Cocoox and Atollic TrueStudio seemed very similar, so the choice fell on Atollic TrueStudio, which has very few limitations. It was easy to download and install the Atollic TrueStudio IDE, and it was straight forward to create a new project with the right settings to create a binary file customized to the Discovery board.

9.2 Hardware Abstraction Layer (HAL)

The Atollic TrueStudio IDE also comes with a HAL (Hardware Abstraction Layer) for the Discovery STM32F4. This means that it is fairly easy to set up the IO pins to do as you like. All the drivers are generated and included when the project is created. The Discovery STM32F4 datasheet [1] tells you which timers and which IO ports to activate to start e.g. a USART communication port or PWM programming.

9.3 The Main Structure of the Software

The main purpose of the program is to control the PWM motors, but it also does other things. First of all, it starts to blink the onboard LED lights in a circular motion to signal that the board is connected to the power source and that the program is running. If the blinking pattern stops, it would mean that the program has bugged.

The program also sends a message through the serial communication line to the computer, via the radio telemetry antenna, with the status of the robot every 500ms. This was mostly used during the testing and debugging phases to print out different variables, and to check that the PWM module was working. The program can also receive messages through the same serial communication line, which was also used a lot during the development, to change the angles of the motors in real time, or to set the endpoints for legs, to see that the mathematics in the software was correct.

Serial communication is also used to receive the PWM signals from the radio remote controller. The PWM signals are sent to an Arduino Nano microcontroller, which translates the PWM signals to a text string containing the duty cycles from all the channels of the controller, and forwarded to the Discovery STM32F4 via USART communication. These messages are then being interpreted and used to control the legs.

The leg control module is the most important, and most frequently run task of the program. It uses the math described in chapter 7, Leg Control, and in chapter 8, Walking Algorithm to calculate the desired motor angles of all the motors on the robot.

The only piece of code that is not written for the Discovery STM32F4 is a small piece of code written for the Arduino Nano, which interprets the PWM signals from the radio remote controller. It measures the length of the duty cycle of all four channels and sends it as a text string via serial communication to the Discovery STM32F4. The software for the Arduino is only 27 lines of code, and it can be found in appendix C.8.

9.4 Virtual Threading

One of the features that are usually available in the C++ language, but that was not possible to use on the Discovery STM32F4, was threading. When this was discovered, some research was done on implementing an RTOS (Real Time Operating System)

on the Discovery, but the conclusion was that it was easier to implement a simpler form of threading by using a timer interrupt to create a countdown for each "thread". The HAL has a timer interrupt that is set to interrupt every 1ms. Each time the interrupt is called, the function SysTick_Handler() from the HAL's stm32fxx_it.c file is called. The function can be found in appendix C.2.

The main.cpp file is the other component of the virtual threading. The main file consists of a while loop that checks if any of the counter variables for the different "threads" have reached zero. If so, the respective function will be called, and the countdown will be reset. A piece of the main.cpp file can be seen below.

```
int main(void)
{
  // SysTick Configuration: Subtracts 1 from all the timers each ms.
  SysTick_Config(SystemCoreClock/1000);
  InitTimers();


  SharedDataModule SharedData;
  Leds leds;
  CommunicationModule comm(SharedData);
  MotorControlModule motorControl(SharedData);

  while{1}
  {
        if(LedDelay == 0)
    {
          leds.ToggleLeds();
          LedDelay = LED_DELAY;
    }
    if(CommunicationModuleDelay == 0)
    {
      comm.runCommunicationModule();
      CommunicationModuleDelay = COMMUNICATION_MODULE_DELAY;
    }
    if(CommunicationModuleReadDelay == 0)
    {
      comm.readPwmInput();
      CommunicationModuleReadDelay = COMMUNICATION_MODULE_READ_DELAY;
    }
    if(MotorControlModuleDelay == 0)
```

```
      {
        motorControl.runMotorControlModule();
        MotorControlModuleDelay = MOTOR_CONTROL_MODULE_DELAY;
      }
    }
}
```

The threading would not be necessary if all the modules could run with the same frequencies, but that is not the case. There is no need to run the communication module as often as the motor control module because it would be impossible to read in real time with the human eye. The same goes with the LED-module. The constants used in the main.cpp file, such as the COMMUNICATION_ MODULE_ DELAY are "const uint32_ t" variables. The values of the delay constants are 500(ms) for the COMMUNICATION_ MODULE_ DELAY, and for the LED_ DELAY, while it is only 10(ms) for MOTOR_ CONTROL_ MODULE_ DELAY.

9.5 Communication Between Different Electronic Hardware Components

The communication between the different hardware components in this project goes via serial communication, or USART (Universal Synchronous/Asynchronous Receiver/Transmitter). USART is not the fastest protocol for transmitting data, but it is easy to set up and does not take much processing capacity. Also, the amount of data sent and received between the units in the robot is fairly low, so USART works well in this case. There is not made a program for a computer to communicate with the robot. All communication is via a serial data transmitter/receiver program, e.g. Termite [6]. The USART module is set up on the Discovery STM32F4 by using some of the functions included in the HAL from Atollic TrueStidio. Some of the variables that have to be set in the software are the baud rate, which is set to 9600[b/second], and the priority of the interrupts. The software on the Discovery motherboard consists of two different serial communication lines. One for communication with the computer, and one for communication with the Arduino that sends the radio controller input. For more information on how USART communication works, see chapter 2.2.

To see how the timers for the USART are set up in the software, and how the different channels are assigned to the IO pins, see appendix C.5.2.

9.6 Shared Data

The program has a lot of data that has to be accessed from different files. Most of the values are created and used by the Leg control module, but some of the data come from the environment, such as through messages via the serial communication line from the computer, and from the PWM input signals from the radio controller. The data input from the computer is not very essential now that the robot is done, but the data from the radio controller is crucial. If someone is to continue working with the robot, other data, such as sensor readings will also be important to send and read between different modules in the software. This means the shared data module will be even more important. Figure 9.1 shows how the different modules all are connected to the shared data module.

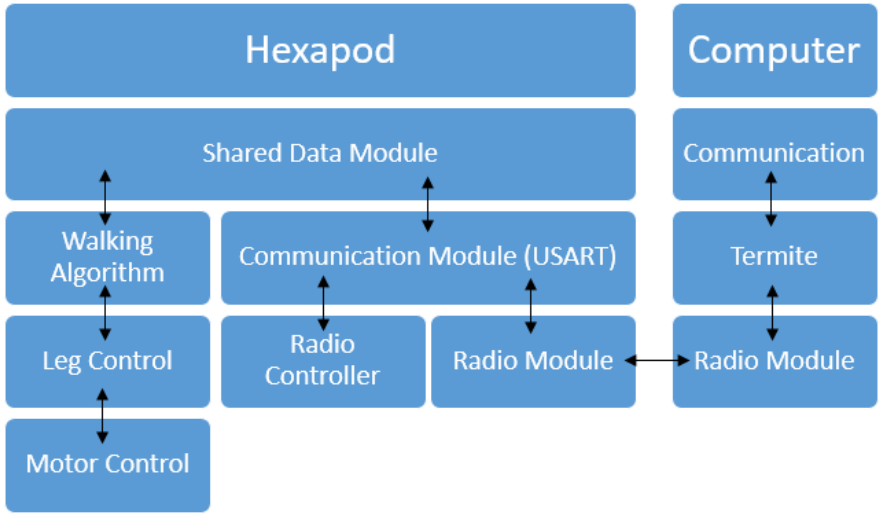

Figure 9.1: Software Layout

The shared data module is initialized in the main function and is passed on as an input to the other modules that need access to the shared data. They then use a pointer to the shared data object to include it in their own class in their initialization. The data can then easily be accessed via a couple of lines of code.

It is important that the different modules do not try to access the shared data at the same time. Therefore a semaphore is used to ensure that only one instance

can access the shared data at the time. It should not be a problem, in theory, since the modules are rarely interrupted. First, they were only interrupted when a serial message was coming in from the computer, but when the signals from the radio controller were introduced, these interruptions occur four times per second. The semaphore is a simple function written in the shared data class, as seen below.

```
bool SharedDataModule::lock()
{
  if(this->m_semaphore)
  {
    this->m_semaphore = false;
    return true;
  }
  else
  {
    return false;
  }
}

bool SharedDataModule::unlock()
{
  if(!this->m_semaphore)
  {
    this->m_semaphore = true;
    return true;
  }
  else
  {
    return false;
  }
}
```

The shared data is then accessed by any class that has included the shared data module by using the following lines of code:

```
if(this->m_pSharedData->lock())
{
  int foo = this->m_pSharedData->value1;
  this->m_pSharedData->unlock();
}
```

9.7 PWM Programming

The PWM control code in the Discovery is also from the HAL. There are some variables that must be set for the PWM control to work with our PWM motors. The PWM module is a counter that counts from zero up to a given value, called the "count register" or $TIMx\_CNT$, before it starts over. The output pulse is set to HIGH when the counter reaches the "compare value", or $TIMx\_CCRx$ (Count Compare Register x), and it stays high until the counter reaches the count register. See figure 9.2. The x'es in "TIMx" refer to what timer that is being set up. The x in "CCRx" refers to what channel, or to what pin, that is used on that timer. Most of the timers have four channels, which means to get 18 PWM outputs, five timers are needed. Luckily. There are four PWM timers that have four outputs channels each on the Discovery STM32F4. The rest of the timers have only one or two output channels. To achieve the 18 PWM outputs that are needed. All the four timers with four PWM outputs are used, in addition to one timer with two PWM outputs.

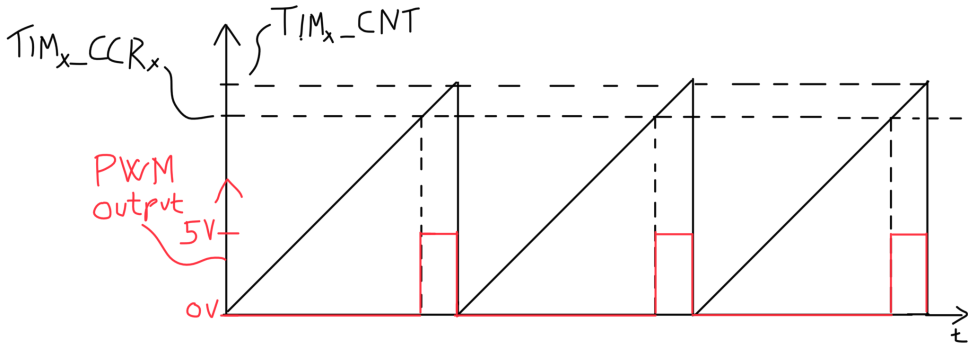

Figure 9.2: PWM Counter

To set the correct period, we must consider what counter interval we want to use. We know that our PWM motors use a 20ms period. For precision, it is desirable to be able to control on a μs level. $20ms = 20000\mu s$. Therefore the count register, $TIMx\_CNT = 19999$. That is 20 000 - 1, since the counter starts on 0.

There are some variations on the different timers used for the PWM outputs. To explain how the values for the initialization for the timers are found, we will use the characteristics of timer 2. See table 9.1 for the values for all the different timers.

Most of the clocks on the timers on the Discovery runs at 84MHz. We want the PWM counter to count to 19999 (20 000 - 1). If we want the period to be exactly 20 000μs, we need to decide on how many ticks on the clock to go by before incrementing the counter. The frequency needed for the timer is the value of the count register times the frequency of the time interval, 50 times per second.

$$Timer\_Frequency = 20000 \cdot 50Hz = 1000000Hz = 1MHz \tag{9.1}$$

The prescaler variable decides how many ticks on the timer clock that is needed for one incrementation on the counter.

$$Timer\_Frequency = \frac{84MHz}{Prescaler} = 1MHz \tag{9.2}$$

As we see in equation 9.2, if the prescaler is chosen to be 84, the counter will have a frequency of 1MHz, which is what we need.

Below is a list of the different timers used, and the different timer values used to achieve a 20 μs PWM period.

Timer nr.	Timer frequency	Prescaler	Count Register	Number of Channels
2	84MHz	84	19999	4
3	84MHz	84	19999	4
4	84MHz	84	19999	4
5	164MHz	164	19999	4
9	84MHz	84	19999	2

Table 9.1: Values for initialization of the timers for PWM outputs

To see how the PWM timers are initialized and how their channels are assigned to the IO pins, see appendix C.6.2.

9.8 Walking Algorithm

The walking algorithm is programmed pretty much the same way as described in chapter 8. The algorithm creates the triangle from figure 8.2 in chapter 8.1. It uses a counter to traverse the lines between the endpoints of the triangle and calculates the next endpoint for each leg. It uses the variables steeringAngle and speed (walking speed) from the Shared Data Module to scale up or down the magnitude of the triangle and to rotate it to make the robot walk in the right direction. The code for the function can be found in appendix C.7.2. See function "void LegControl::UpdateEndpoints(int i), as this acts as the walking algorithm.

9.9 Leg Control

The Leg Control Class has a variable named "double **m_LegPosition". It is a pointer to a pointer to a double variable m_LegPosition[i][j] which corresponds to a point in the triangle discussed in chapter 8.1. As seen in figure 8.2. "i" refers to the point number, and "j" refers to what coordinate (X, Y or Z) it is. The leg control function runs through once for each leg and uses the endpoints found in the walking algorithm to calculate the motor angles for each motor in that leg by using the math found in chapter 7. The code can be seen in appendix C.7.2. See function "void LegControl::CalculateMotorPositions(int i), as this acts as the leg controller.

Chapter 10
Further Work

Since the robot needs some modifications to work properly, it might suit as a future master's thesis for another student. Besides changing the motors, this chapter will give some ideas on what features that could be added to the hexapod.

10.1 Acrobatic Mode

If the robot is going to be used for promotion of the engineering studies at upper secondary schools (VGS in Norwegian) in the future, it would be good to have an extra show off factor. An example is to rotate the body, lean to the sides, or to tilt it using the joystick that is not used for walking. An example is shown in the start of this Youtube video: https://www.youtube.com/watch?v=rAeQn5QnyXo

10.2 Positioning System and Autonomous Control

If an autonomous controller is implemented, the robot would need a positioning system. Using a GPS is one option, but it is not very accurate. A standard GPS has an error of some meters, which is a lot when the robot itself is less than a meter wide. One solution is to use a differential GPS system. They are more accurate, as they use one GPS receiver on a stationary base and one GPS receiver on the robot. The GPS module then calculates the difference in position and gives the relative position of the robot. These solutions are very accurate, as they give the relative position with an error as low as a couple of centimeters. However, they are quite expensive. Other local positioning equipment should also be considered.

Software written for a computer could also be made, which communicates with the robot, sending messages through the telemetry link. The program could give

instructions to start or stop walking, and display the robot's position on a map in real time.

10.3 Anti-Collision System

If an autonomous controller is implemented, there should be an anti-collision system. This is needed because the positioning system might not be very accurate. The robot should have distance sensors to see the distance to walls etc. Also, it should respond to obstacles that are not stationary, such as humans, furniture or other obstacles.

10.4 Automatic Wireless Charging

Assuming one has a positioning system and an autonomous controller implemented, the robot could walk around on its own. A feature that would make it even more autonomous is if it would charge itself, e.g. by walking to a charging station and charge using induction. This way, the robot could walk for days, or weeks, without interacting with humans.

10.5 Camera with live feed

A feature that would be nice to have, although not necessary, is a camera on top of the robot with a live feed to a computer, or even better, a live feed to VR goggles. The "best" solution would be a camera on top of a motor that could rotate the camera according to the motion of the VR goggles. This way the robot could change the forward direction to whatever direction the person is looking so that when walking forwards with the controller, it would always be the direction that is forward in the VR goggles.

Chapter 11
Discussion

11.1 Insufficient Torque In The Motors

The motors that were ordered were not strong enough to lift the body of the robot off of the ground. However, the project was finished to produce a working hexapod software.

The reason why the motors were not strong enough may be many. The motors were ordered from a not so reliable online site, and thus might be cheap copies that do not satisfy the specifications listed in the datasheet. Another reason might be that the total weight of the hardware was heavier than expected. Most of the specifications of the components were known when the calculations were done in the pre-project, but some were unknown, and some of the components were added later on. Besides, the weight of the mass density of the 3D-prints was not known. The last possibility is that there were some errors in the calculations done in the pre-project, which led to wrong torque requirements for the robot.

This problem was discovered late this semester, as the robot required almost finished source code to power all the motors at the same time. The motors will not hold any position without having an input PWM signal. While it is not very difficult to fix, either by ordering new motors or 3D-printing shorter legs it was not enough time to do so before the submission date of this report.

To work around the motor problem while the robot was being tested, it was placed on top of a little box to see that all the legs moved as they were supposed to, and the walking algorithm seems to work fine. A video of a demonstration of the robot is attached to this thesis. It can also be found on Youtube:
https://www.youtube.com/watch?v=KO0w32_NCcQ&feature=youtu.be

11.2 Reliability Of The Software

It has been verified that the robot responds correctly to the radio controller input and that it can walk in all directions. It has also been set to walk and left walking for more than 15 minutes with no problems. This suggests that the semaphores that are implemented are working properly, preventing the software from freezing, or doing unexpected movements. The semaphores keep the different modules from accessing the shared variables at the same time. For higher efficiency, one could have used different semaphores for different types of variables, but during regular use, only the radio hand controller uses interrupt programming, which is only four times per second, and it does not seem to be a problem.

11.3 3D-Printed Parts

The parts designed in SolidWorks seem to be working as they intended. The different mechanisms for attaching the motors to the joints is working as expected, and the printed parts are more than strong enough, which was a concern before actually testing them, as there was little knowledge of how strong the plastic used in the 3D-printer was. If anything is to be changed, it would be the attachment from the motors to the legs. The innermost motor in each leg is attached on both sides of the rotational axis of the motor and seems to be more than strong enough. The rest of the motors are only attached on one side of the motors, the side that the rotors come out. Of course, adding an attachment on the other side as well would increase the amount of plastic used, and thus increase the weight of the robot in total. The connections, as they are now, seems to be working fine. However, to increase the robustness of the robot, this could be given one more round of consideration.

11.4 USART Communication

The communication protocol used between the electronic components in this project, USART, is not the fastest way of communicating, and it needs one separate connection line between each of the components. However, the amount of data transmitted is quite low, so the USART communication seems to be sufficient. Other communication protocols, such as I2C, could have been used. However, they would not work with the radio link used in this project. Of course, there are other ways of communicating that can send more data if that is needed for e.g. a camera feed. This would require a communication protocol with a higher bandwidth. Using a Wifi module, and implement a socket connection with a computer could be the solution for this, but for now, the USART ports work just fine.

11.5 Future Modifications of the Software

The main task of this project was to make a robot that could walk in all directions, which was controlled by a handheld radio controller. This has been done, and the robot is reading the direction of the joysticks on the radio controller, and translating it into a walking speed variable and a walking direction variable. This means that it is easy for someone who has read this report and has access to the source code to modify the software to doing more complex tasks, such as implementing an autonomous controller. All that has to be done is change the way that the walking speed and walking direction variables are controlled.

Chapter 12
Conclusion

The robot is now done, and the legs move satisfyingly in a walking pattern. The robot is also controllable via a handheld radio controller, and it can walk in all directions by changing what end that acts as the front end, according to the problem description. The robot is also quite modifiable, if someone wants to implement an autonomous controller, or adding more sensors or features in the future.

As mentioned earlier in the report, the torque of the motors are not sufficient to lift the robot off of the ground, which means the robot can not walk, but the rest of the hardware and software work properly.

All in all the robot is doing what it is supposed to do, and the software is reliable as it does not freeze, whatever input is applied by either the radio controller or the USART serial communication line.

I hope that someone will continue the work on this robot in the future, to fix the motor problems, and maybe also add some new features, either some of the ones that are mentioned in chapter 10, or some that have not yet been considered. This way, there is still possible that Kybulf jr. some day will be put to use, promoting the engineering studies at NTNU.

References

[1] ST Microelectronics,
STM32F407xx Microcontroller Datasheet,
4th revision, 2013

[2] Electricks
http://www.elektricks.net/wp-content/uploads/2015/02/STM32F4-Discovery.jpg
07.06.2016

[3] Tower Pro
http://www.micropik.com/PDF/SG90Servo.pdf
07.06.2016

[4] Robotplatform
http://www.robotplatform.com/knowledge/servo/servo_control_tutorial.html
07.06.2016

[5] Flysky FS-T4B Radio Controller
http://www.flysky-cn.com/products_detail/productId=39.html
04.01.2017

[6] Termite
http://www.compuphase.com/software_termite.htm
04.01.2017

[7] 3DRRadio
http://img.banggood.com/thumb/water/upload/2015/06/SKU248598-9.jpg
04.01.2017

[8] 3DR Radio Config software
http://ardupilot.org/copter/docs/common-downloads_advanced_user_tools.html
04.01.2017

[9] Arduino Nano
http://brantelonline.com/image/catalog/Arduino%20Nano%20(Original)/3.png
04.01.2017

[10] Bronto Rx Pack Battery
http://www.elefun.no/p/prod.aspx?v=15855
04.01.2017

[11] Clas Ohlson Power Bank
http://www.clasohlson.com/no/Clas-Ohlson,-Powerbank-2600-mAh-/Pr386941000
04.01.2017

[12] Mark W. Spong, Seth Hutchinson, and M. Vidyasagar
Robot Modeling and Control
John Wiley & Sons, INC. 1st Edition 2005

[13] Law of Cosines Illustration
http://mathworld.wolfram.com/LawofCosines.html
07.06.2016

[14] Pinout Overview of Discovery STM32F4
http://stm32f4-discovery.com/wp-content/uploads/2014/06/stm32f4_discovery1.jpg
11.01.2017

[15] SolidWorks Model of TowerPro 9g Servo Motor
https://grabcad.com/library/micro-servo-9g-sg90-tower-pro-1
11.01.2017

[16] Figure of Rotation Around the Z-axis
http://www.neilmendoza.com/wp-content/uploads/2013/01/Rotation_about_z-axis.png
11.01.2017

[17] USART Communication
https://en.wikipedia.org/wiki/Universal_asynchronous_receiver/transmitter#/media/File:UART_timing_diagram.svg
11.01.2017

Chapter A
Communication Protocol

A.1 Communication Protocol

This is a description of the main USART serial channel. On the Discovery STM32F4 board, the RX pin is PA2, and the TX pin is PA3. Every message that is sent starts with "$xx,messageData;'0xA' ", where xx is the MessageID and messageData is the payload data. All the messages is ended with a '0xA'(linefeed).

The Communication Protocol is easily expanded to adapt to data we need to send. Note that in the table %d stands for an int variable, and %f stands for a float variable.

MessageID	Payload	Value Description	Explanation
01	%d,%d	PWM1, PWM2, PWM3	Used to change the PWM motor angles(for testing)
02	%d,%d	Angle1, Angle2	Used to set angles of servomotors
03	%f, %f, %f	X, Y, Z	Used to set the coordinates of the endpoint of a leg. (Used for testing during development)
04	%f	steeringAngle	Used to set the steering angle of the robot (Can be used to control robot without radio controller)
05	%f	walkingSpeed	Used to set the walking speed of the robot (Can be used to control robot without radio controller)
06	-	-	-
50	%d,%d,%d	PWM1, PWM2, PWM3	Used to send current PWM values of leg 1
51	%f,%f,%f	endX,endY,endZ	Used to send the current endpoint values of leg 1
52	%f,%f,%f,%f	WalkingCounter, walkingSpeed, WalkingAngle Elevation	Sends the current walkingCounter (step # in the walking triangle), walking speed, walking angle and elevation of the robot
53	-	-	-

Table A.1: Comparing the Motor Alternatives

Chapter B
CAD Drawings

This appendix contains the CAD drawings for the parts that were designed in SolidWorks. The drawings show measurements that are needed to recreate the parts. The parts are also included as SolidWorks parts in the zip folder attached to this thesis. Note that some of the measurements for the Middle Leg and the Outer Leg are hard to show in the cad files. The important measurements for these parts are the distances between the axis of rotation, which are 100 and 150 mm for the middle leg and the outer leg respectively, in addition to the measurements of the indents used to attach them to the motors. The rest of the measurements for those parts are mostly cosmetic.

The CAD drawing for the inner leg in figure B.4 lack some of the measurements. That is because if all the measurements are included the drawing would be very messy. The inner leg consists of two parts that are the same as the outer motor socket, and the measurements for each motor socket in the inner leg are the same as the measurements in figure B.3

All the measurements given in the CAD drawings are given in mm.

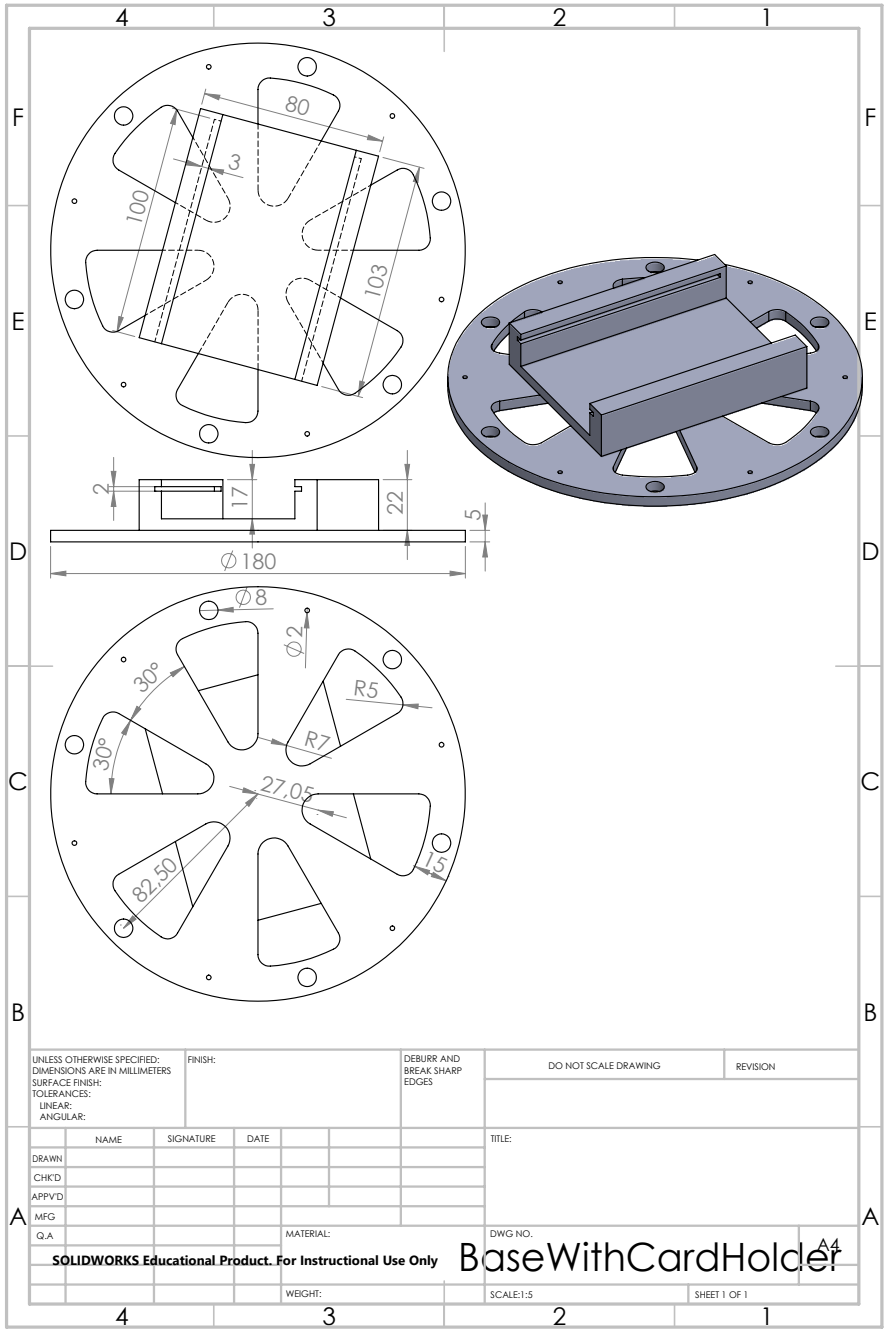

Figure B.1: Cad Drawing of Lower Base with Card Holder

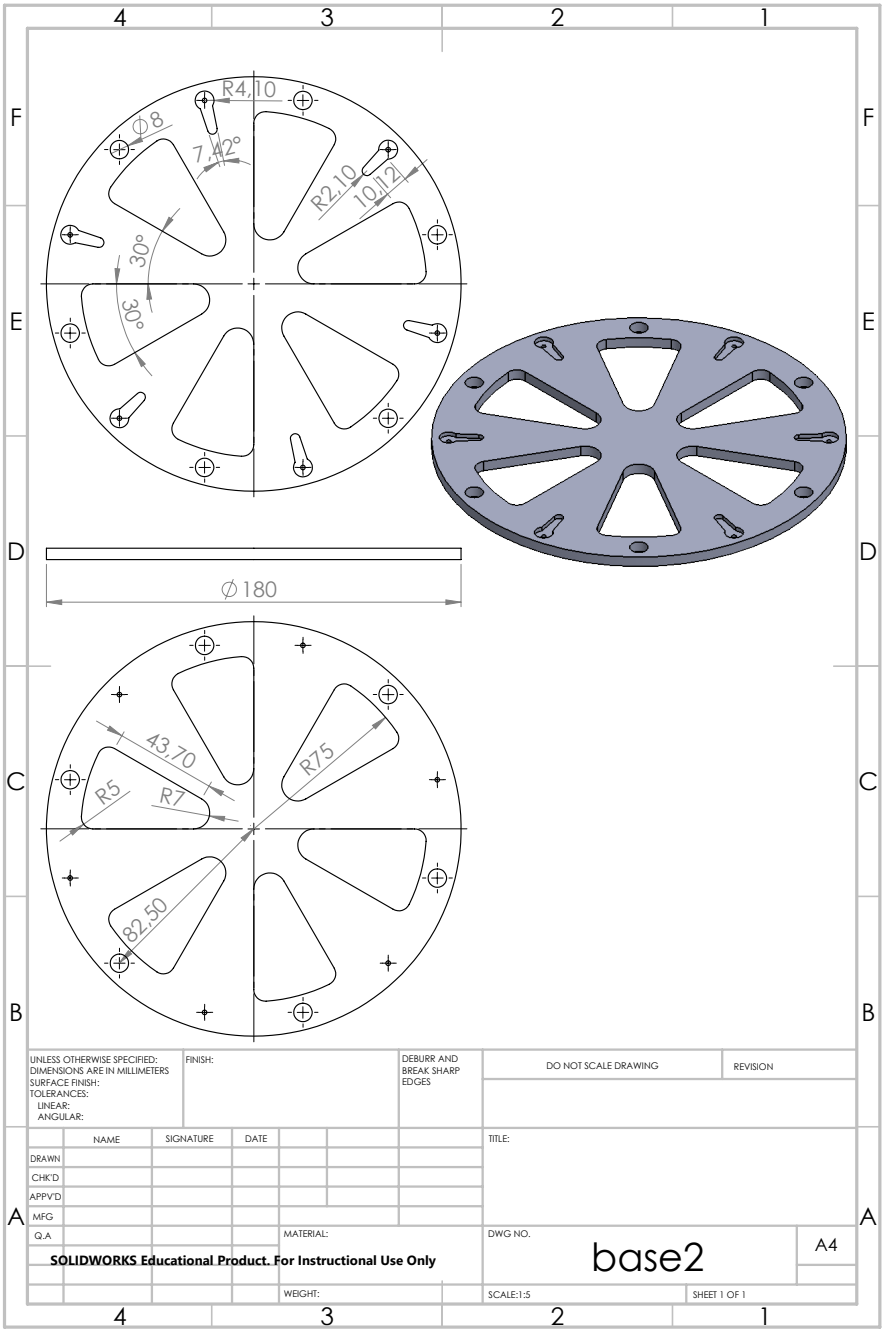

Figure B.2: Cad Drawing of Upper Base

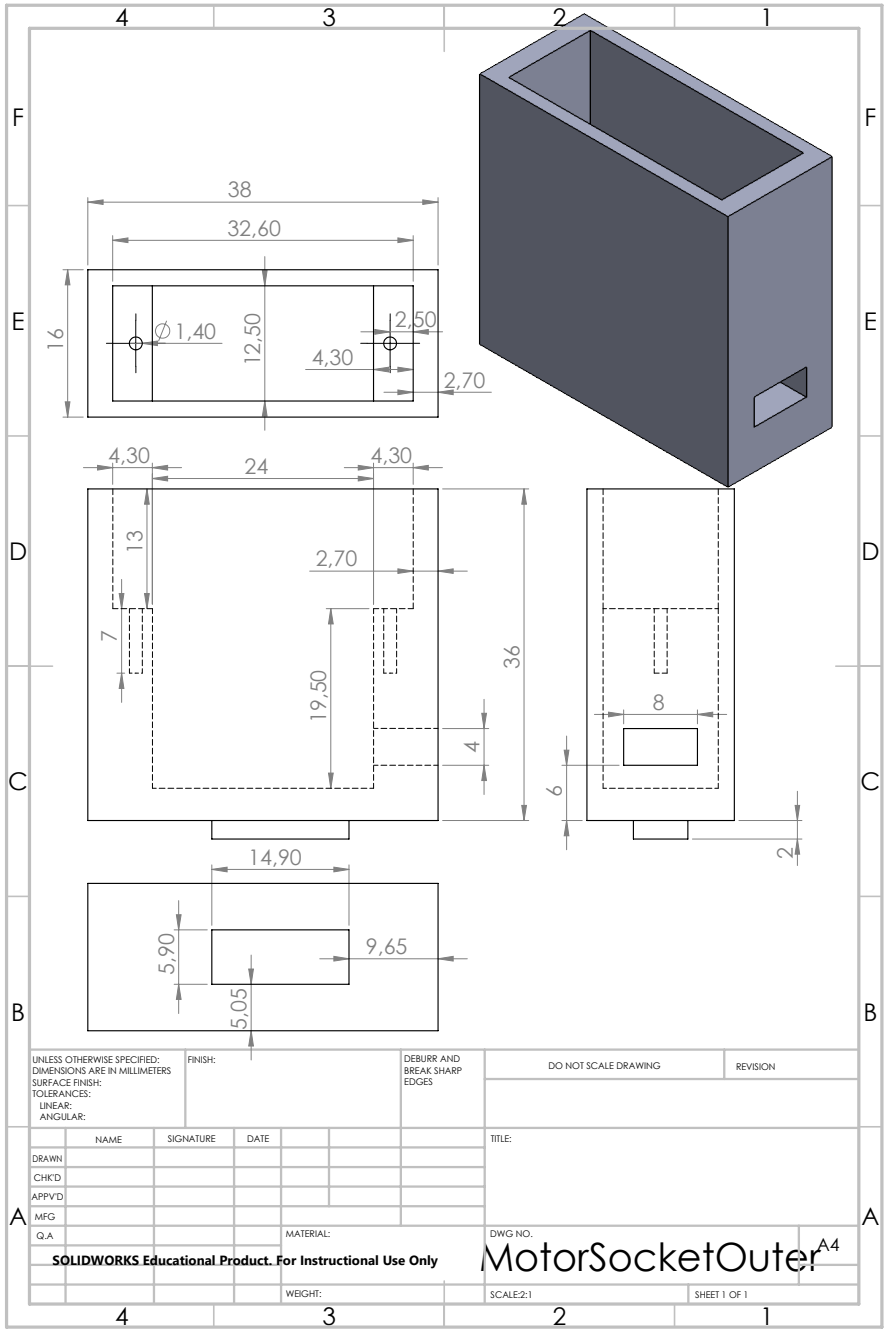

Figure B.3: Cad Drawing of the Outer Motor Socket

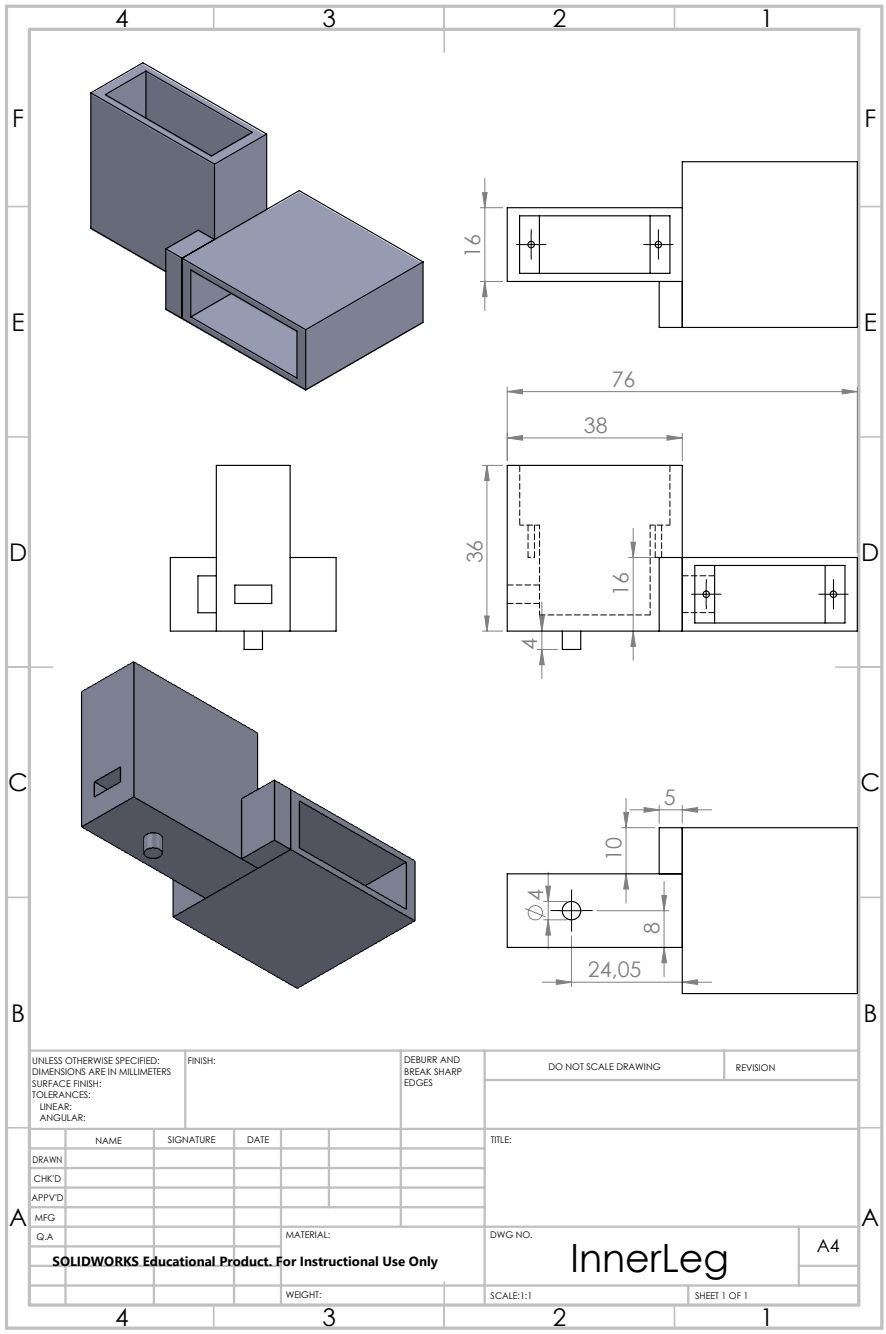

Figure B.4: Cad Drawing of the Inner Leg

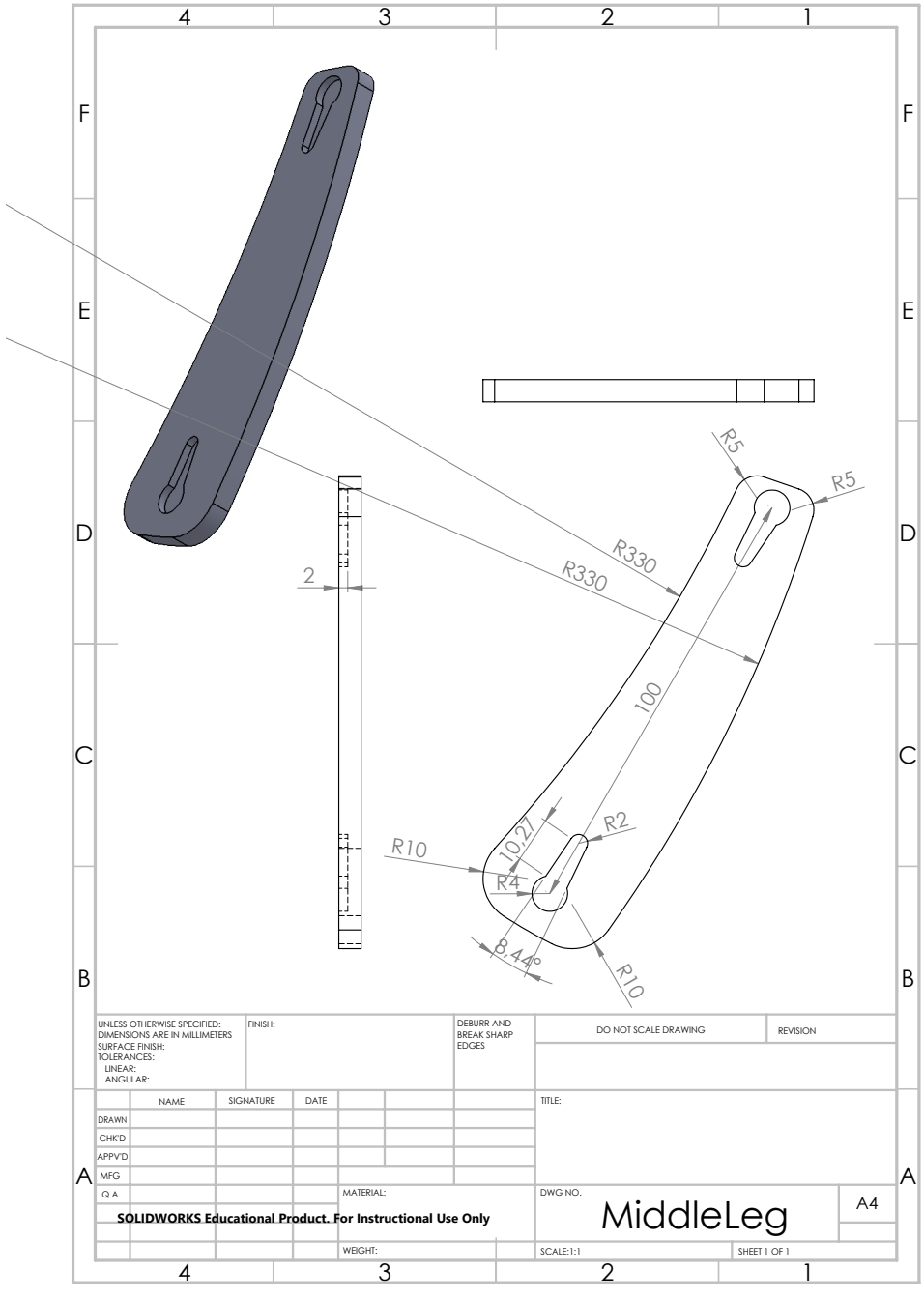

Figure B.5: Cad Drawing of the Middle Leg Part

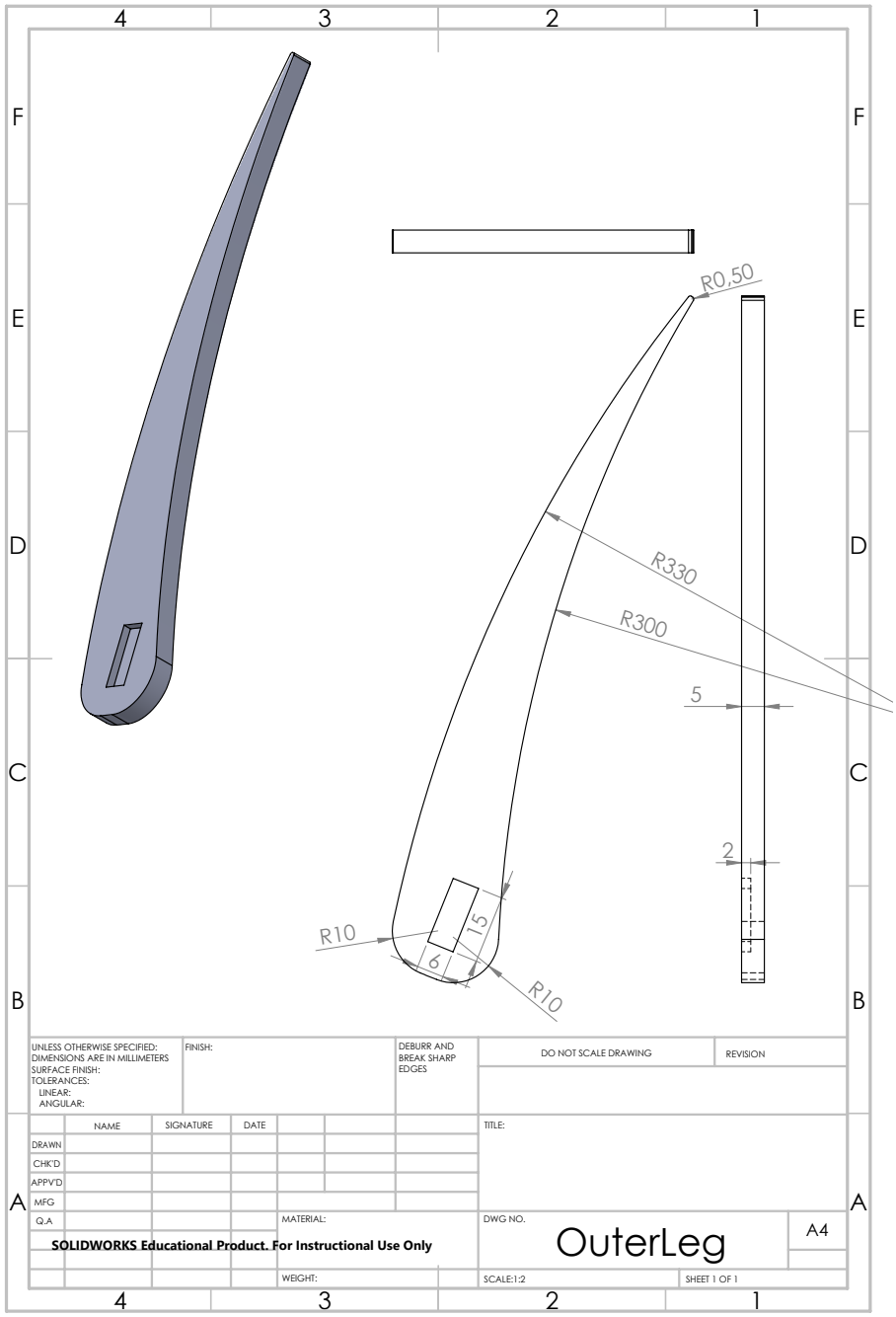

Figure B.6: Cad Drawing of the Outer Leg Part

Chapter C
Software

This appendix contains some of the software developed for this project.

C.1 Main.cpp

Main.cpp acts as the replacement of the real-time operative system. It uses the *SysTick_Handler*() function in section C.2 that is found in *src/stm32f4xx_it.c* to decide how often each of the modules are running.

```
/* Includes */
#include <stdio.h>
#include "stm32f4xx.h"
#include "stm32f4xx_usart.h"
#include "Led.h"
#include "CommunicationModule.h"
#include "MotorControlModule.h"
#include "SharedDataModule.h"
#include <math.h>

extern __IO uint32_t TimingDelay;
extern __IO uint32_t LedDelay;
extern __IO uint32_t CommunicationModuleDelay;
extern __IO uint32_t CommunicationModuleReadDelay;
extern __IO uint32_t MotorControlModuleDelay;

const uint32_t LED_DELAY = 500;
const uint32_t COMMUNICATION_MODULE_DELAY = 500;
const uint32_t COMMUNICATION_MODULE_READ_DELAY = 1;
const uint32_t MOTOR_CONTROL_MODULE_DELAY = 10;

void InitTimers(void);

int main(void)
```

```
{
  //SysTick Configuration: Subtracts 1 from all the timers each
      ms.
  SysTick_Config(SystemCoreClock/1000);
  InitTimers();

  SharedDataModule SharedData;
  Leds leds;
  CommunicationModule comm(SharedData);
  MotorControlModule motorControl(SharedData);

  while(1)
  {
    if(LedDelay == 0)
    {
      leds.ToggleLeds();
      LedDelay = LED_DELAY;
    }
    if(CommunicationModuleDelay == 0)
    {
      comm.runCommunicationModule();
      CommunicationModuleDelay = COMMUNICATION_MODULE_DELAY;
    }
    if(CommunicationModuleReadDelay == 0)
    {
      comm.readPwmInput();
      CommunicationModuleReadDelay = 1;
    }
    if(MotorControlModuleDelay == 0)
    {
      motorControl.runMotorControlModule();
      MotorControlModuleDelay = MOTOR_CONTROL_MODULE_DELAY;
    }
  }


  return 0;
}

void InitTimers(void)
{
  LedDelay = LED_DELAY;
  CommunicationModuleDelay = COMMUNICATION_MODULE_DELAY;
  MotorControlModuleDelay = MOTOR_CONTROL_MODULE_DELAY;
}
```

C.2 SystickHandler

This is a function from the file *src/stm32f4xx_it.c*.

```
void SysTick_Handler(void)
{
  /*  TimingDelay_Decrement(); */
  if(TimingDelay > 0)
  {
  TimingDelay--;
  }
  if(LedDelay > 0)
  {
  LedDelay--;
  }
  if(CommunicationModuleDelay > 0)
  {
  CommunicationModuleDelay--;
  }
  if(MotorControlModuleDelay > 0)
  {
  MotorControlModuleDelay--;
  }
}
```

C.3 Usart communication handling

Note that this function only saves the messaged received in the volatile "MyMessage-Buffer" variable. Another program checks the "IncommingMessage" variable, and if it is 1, it reads the buffer and sets "IncomingMessage" to 0. This function is also part of the *src/stm32f4xx_it.c* file, as it is the file that handles all the interruptions.

```
void USART2_IRQHandler(void)
{
  // Check the Interrupt status to ensure the Rx interrupt was
      triggered, not Tx
  if( USART_GetITStatus(USART2, USART_IT_RXNE))
  {
    static int cnt = 0;
    // Get the byte that was transferred
    char ch = USART2->DR;

    // Check for "Enter" key, or Maximum characters
    if((ch != '\n') && (cnt < MAX_BUFFER_LENGTH))
    {
      MyMessageBuffer[cnt++] = ch;
    }
    else
    {
      MyMessageBuffer[cnt] = '\n';
      cnt = 0;
      IncommingMessage = 1;
      CommunicationModuleDelay = 0; // To read the message right
          away

    }
  }
}
```

C.4 Shared Data Module

Some of the variables in the shared data module are removed from the script below. That is because they are not used in the software in the walking algorithm, but they were used during the software development. They are still in the source code in the zip folder.

C.4.1 SharedDataModule.h

```
#ifndef SHAREDDATAMODULE_H_
#define SHAREDDATAMODULE_H_

#include "stm32f4xx.h"

class SharedDataModule
{

public:
  SharedDataModule();
  ~SharedDataModule();
  bool lock();
  bool unlock();

  bool m_changes;
  uint32_t m_Pwm1Output;
  uint32_t m_Pwm2Output;
  uint32_t m_Pwm3Output;
  double m_endX;
  double m_endY;
  double m_endZ;
  uint32_t *m_PwmIn;
  double m_steeringAngle;
  double m_speed;
  double m_elevationRate;
  double m_elevation;
  uint32_t m_walkingCounter;

private:
  bool m_semaphore;
};

extern SharedDataModule SharedData;
#endif /* SHAREDDATAMODULE_H_ */
```

C.4.2 SharedDataModule.cpp

```
#include "SharedDataModule.h"


SharedDataModule::SharedDataModule()
:m_changes(false),
 m_Pwm1Output(1250),
 m_Pwm2Output(1250),
 m_Pwm3Output(1250),
 m_endX(20),
 m_endY(0),
 m_endZ(-15),
 m_steeringAngle(0),
 m_speed(0),
 m_elevationRate(0),
 m_elevation(0),
 m_walkingCounter(0),
 m_semaphore(true)
{
  m_PwmIn = new uint32_t[4];
  for(int i=0; i<4; i++)
  {
    m_PwmIn[i] = uint32_t(1250);
  }
}

SharedDataModule::~SharedDataModule()
{

}

bool SharedDataModule::lock()
{
  if(this->m_semaphore)
  {
    this->m_semaphore = false;
    return true;
  }
  else
  {
    return false;
  }
}

bool SharedDataModule::unlock()
```

```
{
  if(!this->m_semaphore)
  {
    this->m_semaphore = true;
    return true;
  }
  else
  {
    return false;
  }
}
```

C.5 Communication Module

The communication module is also a bit simplified. Only the code used to initialize one of the USART ports are included below. Both are of course included in the file attached in the zip-folder. Also, most of the functions used in *voidCommunicationModule* :: *sendUsartStatus*() are excluded, as they all look very similar to the first one, which is included here.

C.5.1 CommunicationModule.h

```
#ifndef COMMUNICATIONMODULE_H_
#define COMMUNICATIONMODULE_H_

#include "stm32f4xx.h"
#include "stm32f4xx_usart.h"
#include "SharedDataModule.h"
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include "SharedDataModule.h"

const int MAX_BUFFER_LENGTH = 50;

extern __IO uint32_t IncommingMessage;

class CommunicationModule
{
public:
  CommunicationModule(SharedDataModule &SharedData);
  ~CommunicationModule();
  void runCommunicationModule();
  void readPwmInput();
  void USART_puts(USART_TypeDef *USARTx, volatile char *str);
private:
  GPIO_InitTypeDef  m_GPIO_D_InitStructure;
  GPIO_InitTypeDef  m_GPIO_A_InitStructure;
  USART_InitTypeDef   m_USART_InitStructure;
  NVIC_InitTypeDef  m_NVIC_2_InitStructure;
  NVIC_InitTypeDef  m_NVIC_1_InitStructure;

  SharedDataModule *m_pSharedData;

  uint32_t m_MaxBufferLength;
  char m_receivedBuffer[MAX_BUFFER_LENGTH + 1];
  char m_receivedPWMBuffer[MAX_BUFFER_LENGTH + 1];
```

```
  uint32_t m_messageStart;

  void messageHandler();
  void updateParameters();
  void updatePWMInput();
  void sendUsartStatus();
  void sendPwmStatus();
  void sendEndpointStatus();
  void sendControllerStatus();
  void sendLineBreak();

};



#endif /* COMMUNICATIONMODULE_H_ */
```

C.5.2 CommunicationModule.cpp

```
#include "CommunicationModule.h"
#include <string>
#include <sstream>

extern __IO char MyMessageBuffer[100];
extern __IO char PWMMessageBuffer[100];

double sign(double i);

CommunicationModule::CommunicationModule(SharedDataModule &
    SharedData)
:m_pSharedData(&SharedData),
m_MaxBufferLength(MAX_BUFFER_LENGTH),
m_messageStart(0)
{
  //Enable the periph clock for USART2;
  RCC_APB1PeriphClockCmd(RCC_APB1Periph_USART2, ENABLE);
  //Enable the GPIOD clock;
  RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOD, ENABLE);

  // Setup the GPIO pins  for Tx (PD5) and Rx(PD6)
  m_GPIO_D_InitStructure.GPIO_Pin   = GPIO_Pin_5 | GPIO_Pin_6;
  m_GPIO_D_InitStructure.GPIO_Mode    = GPIO_Mode_AF;
  m_GPIO_D_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
  m_GPIO_D_InitStructure.GPIO_OType = GPIO_OType_PP;
  m_GPIO_D_InitStructure.GPIO_PuPd    = GPIO_PuPd_UP;
  GPIO_Init(GPIOD, &m_GPIO_D_InitStructure);

  // Connect PD5 and PD6 with the USART2 Alternate Function
  GPIO_PinAFConfig(GPIOD, GPIO_PinSource5, GPIO_AF_USART2);
  GPIO_PinAFConfig(GPIOD, GPIO_PinSource6, GPIO_AF_USART2);


  m_USART_InitStructure.USART_BaudRate        = 9600;
  m_USART_InitStructure.USART_WordLength      = USART_WordLength_8b
      ;
  m_USART_InitStructure.USART_StopBits        = USART_StopBits_1;
  m_USART_InitStructure.USART_Parity        = USART_Parity_No;
  m_USART_InitStructure.USART_HardwareFlowControl =
      USART_HardwareFlowControl_None;
  m_USART_InitStructure.USART_Mode          = USART_Mode_Tx |
      USART_Mode_Rx;
  USART_Init(USART2, &m_USART_InitStructure);
```

```
  /* Enable the USART2 receive interrupt and configure
    the interrupt controller to jump to USART2_IRQHandler() if
        the USART2 receive interrupt occurs
  */
  USART_ITConfig(USART2, USART_IT_RXNE, ENABLE);
  /************See stm32f4xx_it.c for interrupt handling
      ************/

  //NVIC_PriorityGroupConfig(NVIC_PriorityGroup_1);
  m_NVIC_2_InitStructure.NVIC_IRQChannel                    = USART2_IRQn
      ;
  m_NVIC_2_InitStructure.NVIC_IRQChannelPreemptionPriority  = 0;
  m_NVIC_2_InitStructure.NVIC_IRQChannelSubPriority         = 1;
  m_NVIC_2_InitStructure.NVIC_IRQChannelCmd                 = ENABLE;
  NVIC_Init(&m_NVIC_2_InitStructure);

  // Finally enable the USART2 peripheral
  USART_Cmd(USART2, ENABLE);
}

void CommunicationModule::USART_puts(USART_TypeDef *USARTx,
    volatile char *str){
  while(*str){
    // Wait for the TC (Transmission Complete) Flag to be set
    // while(!(USARTx->SR & 0x040));
    while(USART_GetFlagStatus(USART2, USART_FLAG_TC) == RESET);
    USART_SendData(USARTx, *str);
    *str++;
  }
}


void CommunicationModule::runCommunicationModule()
{
  this->updateParameters();
  this->sendUsartStatus();
}

void CommunicationModule::updateParameters()
{
if (1 == IncommingMessage)
  {
    //Buffer from USART2:
    static uint32_t cnt = 0;
    for(int i=0; i<MAX_BUFFER_LENGTH+1; i++)
```

```
    {
      m_receivedBuffer[i] = '\0';
    }
    // Get the byte that was transferred
    char ch = MyMessageBuffer[cnt];

    // Check for "Enter" key, or Maximum characters
    while((ch != '\n') && (cnt < this->m_MaxBufferLength))
    {
      m_receivedBuffer[cnt] = ch;
      cnt++;
      ch = MyMessageBuffer[cnt];
    }
    m_receivedBuffer[cnt] = '\n';
    cnt = 0;
    USART_puts(USART2, m_receivedBuffer);



    IncommingMessage = 0;
    this->messageHandler();
  }
}

void CommunicationModule::messageHandler()
{
  uint32_t messageId = 0;
  uint32_t crc;
  sscanf((char*)&m_receivedBuffer, "$%d,", &messageId);

  switch(messageId)
  {
  case 1:
  {
    //Updating PWM signals
    uint32_t Pwm1;
    uint32_t Pwm2;
    uint32_t Pwm3;
    sscanf((char*)&m_receivedBuffer[4], "%d,%d,%d;", &Pwm1, &Pwm2
        , &Pwm3);
    if(m_pSharedData->lock())
    {
      m_pSharedData->m_Pwm1Output = Pwm1;
      m_pSharedData->m_Pwm2Output = Pwm2;
      m_pSharedData->m_Pwm3Output = Pwm3;
```

```
        m_pSharedData->m_changes = true;
        m_pSharedData->unlock();
      }
      break;
    }
    //case 2:... The rest of the cases are excluded in the appendix
    default:
      break;
    }
}

void CommunicationModule::sendUsartStatus()
{
  this->sendPwmStatus();
  this->sendEndpointStatus();
  this->sendControllerStatus();
  this->sendLineBreak();
}


void CommunicationModule::sendPwmStatus()
{
  int PWM1;
  int PWM2;
  int PWM3;
  if(m_pSharedData->lock())
  {
    PWM1 = (int)m_pSharedData->m_Pwm1Output;
    PWM2 = (int)m_pSharedData->m_Pwm2Output;
    PWM3 = (int)m_pSharedData->m_Pwm3Output;
    m_pSharedData->unlock();
  }

  char buffer[MAX_BUFFER_LENGTH];
  sprintf(buffer, "&50, %i, %i, %i;\n", PWM1, PWM2, PWM3);
  this->USART_puts(USART2, buffer);
}



void CommunicationModule::readPwmInput()
{
  //Buffer from PWM input:
  static uint32_t cnt = 0;
  cnt = 0;
```

```
  for(int i=0; i<MAX_BUFFER_LENGTH+1; i++)
  {
    m_receivedPWMBuffer[i] = '\0';
  }
  // Get the byte that was transferred
  char ch = PWMMessageBuffer[cnt];

  // Check for "Enter" key, or Maximum characters
  while((ch != '\n') && (cnt < this->m_MaxBufferLength))
  {
    m_receivedPWMBuffer[cnt] = ch;
    cnt++;
    ch = PWMMessageBuffer[cnt];
  }
  m_receivedPWMBuffer[cnt] = '\n';
  cnt = 0;

  this->updatePWMInput();

}


void CommunicationModule::updatePWMInput()
{
  int inputChannel;
  int PwmInput[4];
  int b=0;
  for(int i=0; i<4; i++)
  {
    PwmInput[i] = 0;
  }

  sscanf((char*)&m_receivedPWMBuffer, "Ch0:%d.%d,Ch1:%d.%d,Ch2:%d
      .%d,Ch3:%d.%d,", &PwmInput[0], &b, &PwmInput[1], &b, &
      PwmInput[2], &b, &PwmInput[3], &b);

  if(m_pSharedData->lock())
  {
    for(int i=0; i<4; i++)
    {
      if(PwmInput[i] != 0)
      m_pSharedData->m_PwmIn[i] = (uint32_t)PwmInput[i];
    }
    m_pSharedData->unlock();
  }
```

```
}
```

C.6 MotorControl Module

This section is also very simplified in the appendix. The code below shows how to set up two PWM timers, and connect them to pins. A lot of the code that was used during development and tuning is not included here in the appendix. They are included in the attached zip-folder.

C.6.1 MotorControlModule.h

```
#ifndef MOTORCONTROLMODULE_H_
#define MOTORCONTROLMODULE_H_

#include "stm32f4xx.h"
#include "stm32f4xx_usart.h"
#include <math.h>
#include "SharedDataModule.h"

#include "LegControl.h"

const uint32_t PWM_CH0_MIN = 1050;
const uint32_t PWM_CH0_MAX = 1950;
const uint32_t PWM_CH0_AVG = 1500;
const uint32_t PWM_CH1_MIN = 1120;
const uint32_t PWM_CH1_MAX = 1770;
const uint32_t PWM_CH1_AVG = 1445;
const uint32_t PWM_CH2_MIN = 1080;
const uint32_t PWM_CH2_MAX = 1900;
const uint32_t PWM_CH2_AVG = 1490;
const uint32_t PWM_CH3_MIN = 1070;
const uint32_t PWM_CH3_MAX = 1960;
const uint32_t PWM_CH3_AVG = 1515;
const uint32_t PWM_OUT_MAX = 1800;
const uint32_t PWM_OUT_MIN = 700;
const uint32_t PWM_OUT_AVG = 1250;

class MotorControlModule
{
public:
  MotorControlModule(SharedDataModule &SharedData);
  ~MotorControlModule();
  void runMotorControlModule();
private:
  // Structures for configuration
  GPIO_InitTypeDef              m_GPIO_B_InitStructure;
  TIM_TimeBaseInitTypeDef       m_TIM_4_TimeBaseStructure;
  TIM_TimeBaseInitTypeDef    m_TIM_3_TimeBaseStructure;
```

```
TIM_OCInitTypeDef              m_TIM_OCInitStructure;

SharedDataModule *m_pSharedData;

uint32_t m_DutyCycleMax;
uint32_t m_DutyCycleMin;
uint32_t m_Counter;
double m_PI;
int32_t m_Angle1Output;
int32_t m_Angle2Output;
uint32_t m_PWM1Output;
uint32_t m_PWM2Output;
uint32_t m_PWM3Output;
uint32_t *m_PwmInput;
double *m_PwmScalarInput;
double *m_LegAngles;
double m_steeringAngle;
double m_velocity;
double m_speed;
double m_elevationRate;
uint32_t **m_PwmOutput;

uint32_t deg2PWM(double degrees);
uint32_t rad2PwmInnerMotor(double radians);
uint32_t rad2PwmMiddleMotor(double radians);
uint32_t rad2PwmOuterMotor(double radians);
void ScalePwmInputParameters();
void UpdatePwmInput();
void UpdateSpeed();
void UpdateSteeringAngle();
void UpdateElevationRate();

void SetAngle();
void RunSineWave();
void RunTickTack();
void RunLegController();
void UpdatePwmFromInput();

void UpdatePwm();
void UpdateMotors();
void UpdateAngles();
void AdjustAngles();

void UpdateFromLegControl();
LegControl m_leg;
```

```
};



#endif /* MOTORCONTROLMODULE_H_ */
```

C.6.2 MotorControlModule.cpp

```
#include<vector>

#include "MotorControlModule.h"
MotorControlModule::MotorControlModule(SharedDataModule &
    SharedData)
:m_DutyCycleMax(1800),
m_DutyCycleMin(700),
m_Counter(0),
m_PI(3.141592),
m_PWM1Output ( (m_DutyCycleMax + m_DutyCycleMin)/2),
m_PWM2Output ( (m_DutyCycleMax + m_DutyCycleMin)/2),
m_PWM3Output ( (m_DutyCycleMax + m_DutyCycleMin)/2),
m_steeringAngle(0),
m_velocity(0),
m_speed(0),
m_elevationRate(0),
m_leg(*m_pSharedData)
{
  m_PwmInput = new uint32_t[4];
  m_PwmScalarInput = new double[4];
  for(int i=0; i<4; i++)
  {
    m_PwmInput[i] = 1250;
    m_PwmScalarInput[i] = 0;
  }
  m_LegAngles = new double[NUMBER_OF_LEGS];
  m_PwmOutput = new uint32_t*[NUMBER_OF_LEGS];
  for (int i=0; i<NUMBER_OF_LEGS; i++)
  {
    m_LegAngles[i] = 0 + i*2*PI/NUMBER_OF_LEGS;
    m_PwmOutput[i] = new uint32_t[MOTORS_PR_LEG];
    for(int j=0; j<MOTORS_PR_LEG; j++)
    {
      m_PwmOutput[i][j] = 0;
    }
  }
  m_pSharedData = &SharedData;
  // TIM4 Clock Enable
  RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM4, ENABLE);
  // TIM3 Clock Enable
  RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM3, ENABLE);

  // GPIOB Clock Enable
  RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOB, ENABLE);
```

```
// Initalize PB6-9 (TIM4 Ch1-4), PB0-1 (TIM3 Ch3-4), PB3 (TIM2
    Ch2) and PB10-11 (TIM2 Ch3-4)
m_GPIO_B_InitStructure.GPIO_Pin      = GPIO_Pin_6 | GPIO_Pin_7 |
     GPIO_Pin_8 | GPIO_Pin_9 | GPIO_Pin_0 |GPIO_Pin_1 |
    GPIO_Pin_10 | GPIO_Pin_11;
m_GPIO_B_InitStructure.GPIO_Mode     = GPIO_Mode_AF;
m_GPIO_B_InitStructure.GPIO_Speed    = GPIO_Speed_100MHz;      //
    GPIO_High_Speed
m_GPIO_B_InitStructure.GPIO_OType    = GPIO_OType_PP;
m_GPIO_B_InitStructure.GPIO_PuPd     = GPIO_PuPd_UP;           //
    Weak Pull-up for safety during startup
GPIO_Init(GPIOB, &m_GPIO_B_InitStructure);

// Assign Alternate Functions to pins: B
GPIO_PinAFConfig(GPIOB, GPIO_PinSource0, GPIO_AF_TIM3);
GPIO_PinAFConfig(GPIOB, GPIO_PinSource1, GPIO_AF_TIM3);
GPIO_PinAFConfig(GPIOB, GPIO_PinSource3, GPIO_AF_TIM2);
GPIO_PinAFConfig(GPIOB, GPIO_PinSource6, GPIO_AF_TIM4);
GPIO_PinAFConfig(GPIOB, GPIO_PinSource7, GPIO_AF_TIM4);
GPIO_PinAFConfig(GPIOB, GPIO_PinSource8, GPIO_AF_TIM4);
GPIO_PinAFConfig(GPIOB, GPIO_PinSource9, GPIO_AF_TIM4);
GPIO_PinAFConfig(GPIOB, GPIO_PinSource10, GPIO_AF_TIM2);
GPIO_PinAFConfig(GPIOB, GPIO_PinSource11, GPIO_AF_TIM2);

uint16_t PrescalerValue = (uint16_t) 84;

// Time Base Configuration
m_TIM_4_TimeBaseStructure.TIM_Period         = 19999;
m_TIM_4_TimeBaseStructure.TIM_Prescaler      = PrescalerValue;
m_TIM_4_TimeBaseStructure.TIM_ClockDivision = 0;
m_TIM_4_TimeBaseStructure.TIM_CounterMode    =
    TIM_CounterMode_Up;

TIM_TimeBaseInit(TIM4, &m_TIM_4_TimeBaseStructure);

m_TIM_3_TimeBaseStructure.TIM_Period     = 19999;
m_TIM_3_TimeBaseStructure.TIM_Prescaler    = PrescalerValue;
m_TIM_3_TimeBaseStructure.TIM_ClockDivision = 0;
m_TIM_3_TimeBaseStructure.TIM_CounterMode = TIM_CounterMode_Up;

TIM_TimeBaseInit(TIM3, &m_TIM_3_TimeBaseStructure);

m_TIM_2_TimeBaseStructure.TIM_Period     = 19999;
m_TIM_2_TimeBaseStructure.TIM_Prescaler    = PrescalerValue;
```

```
m_TIM_2_TimeBaseStructure.TIM_ClockDivision = 0;
m_TIM_2_TimeBaseStructure.TIM_CounterMode = TIM_CounterMode_Up;

m_TIM_OCInitStructure.TIM_OCMode      = TIM_OCMode_PWM1; //Set
    on compare match
m_TIM_OCInitStructure.TIM_OutputState = TIM_OutputState_Enable;
     //Set pulse when CNT = CCRx
m_TIM_OCInitStructure.TIM_Pulse       = 0;
                            // Initial duty cycle
m_TIM_OCInitStructure.TIM_OCPolarity  = TIM_OCPolarity_High;
      // Active High, i.e. 0 -> 1 starts duty cycle

// TIM4
// Channel 1
TIM_OC1Init(TIM4, &m_TIM_OCInitStructure);
TIM_OC1PreloadConfig(TIM4, TIM_OCPreload_Enable);
// Channel 2
TIM_OC2Init(TIM4, &m_TIM_OCInitStructure);
TIM_OC2PreloadConfig(TIM4, TIM_OCPreload_Enable);
// Channel 3
TIM_OC3Init(TIM4, &m_TIM_OCInitStructure);
TIM_OC3PreloadConfig(TIM4, TIM_OCPreload_Enable);
// Channel 4
TIM_OC4Init(TIM4, &m_TIM_OCInitStructure);
TIM_OC4PreloadConfig(TIM4, TIM_OCPreload_Enable);

TIM_ARRPreloadConfig(TIM4, ENABLE);

// TIM3
// Channel 1
TIM_OC1Init(TIM3, &m_TIM_OCInitStructure);
TIM_OC1PreloadConfig(TIM3, TIM_OCPreload_Enable);
// Channel 2
TIM_OC2Init(TIM3, &m_TIM_OCInitStructure);
TIM_OC2PreloadConfig(TIM3, TIM_OCPreload_Enable);
// Channel 3
TIM_OC3Init(TIM3, &m_TIM_OCInitStructure);
TIM_OC3PreloadConfig(TIM3, TIM_OCPreload_Enable);
// Channel 4
TIM_OC4Init(TIM3, &m_TIM_OCInitStructure);
TIM_OC4PreloadConfig(TIM3, TIM_OCPreload_Enable);
TIM_ARRPreloadConfig(TIM3, ENABLE);


// Start timer:
```

```
  TIM_Cmd(TIM4, ENABLE);
  TIM_Cmd(TIM3, ENABLE);
}

void MotorControlModule::runMotorControlModule()
{
  //this->UpdatePwmFromInput(); // Comment in this to test the
      motors manually.
  this->UpdatePwmInput();
  this->RunLegController();
  this->UpdateMotors();
}

void MotorControlModule::UpdatePwmInput()
{
  if(m_pSharedData->lock())
  {
    for(int i=0; i<4; i++)
    {
      m_PwmInput[i] = m_pSharedData->m_PwmIn[i];
    }

    m_pSharedData->unlock();
  }
  this->ScalePwmInputParameters();
  if(m_pSharedData->lock())
  {
    m_pSharedData->m_steeringAngle = this->m_steeringAngle;
    m_pSharedData->m_speed = this->m_speed;
    m_pSharedData->unlock();
  }
}

void MotorControlModule::ScalePwmInputParameters()
{
  m_PwmScalarInput[0] = (m_PwmInput[0]-((double)PWM_CH0_AVG))/((
      double)PWM_CH0_MAX-(double)PWM_CH0_AVG);
  m_PwmScalarInput[1] = (m_PwmInput[1]-((double)PWM_CH1_AVG))/((
      double)PWM_CH1_MAX-(double)PWM_CH1_AVG);
  m_PwmScalarInput[2] = (m_PwmInput[2]-((double)PWM_CH2_AVG))/((
      double)PWM_CH2_MAX-(double)PWM_CH2_AVG);
  m_PwmScalarInput[3] = (m_PwmInput[3]-((double)PWM_CH3_AVG))/((
      double)PWM_CH3_MAX-(double)PWM_CH3_AVG);

  for(int i=0; i<4; i++)
```

```
  {
    if(m_PwmScalarInput[i] < -1)
    {
      m_PwmScalarInput[i] = -1;
    }
    if(m_PwmScalarInput[i] > 1)
    {
      m_PwmScalarInput[i] = 1;
    }
  }
  this->UpdateSpeed();
  this->UpdateSteeringAngle();
  this->UpdateElevationRate();
  if(m_pSharedData->lock())
  {
    m_pSharedData->m_speed = this->m_speed;
    m_pSharedData->m_steeringAngle = this->m_steeringAngle;
    m_pSharedData->m_elevationRate = this->m_elevationRate;
    m_pSharedData->unlock();
  }
}

void MotorControlModule::UpdateSpeed()
{
  m_speed = sqrt(m_PwmScalarInput[3]*m_PwmScalarInput[3] +
      m_PwmScalarInput[2]*m_PwmScalarInput[2]);
  double tempSpeed = 0;
  for (int i=0; i<6; i++)
  {
    double j = (double)i;
    if(m_speed > j/5)
    {
      tempSpeed = (j/5);
    }
  }
  m_speed = tempSpeed;
}


void MotorControlModule::RunLegController()
{
  this->m_leg.RunLegController();
  for(int i=0; i<NUMBER_OF_LEGS; i++)
  {
```

```
    uint32_t pwmValue = rad2PwmInnerMotor(m_leg.getMotorAngle(i
        ,0));
    pwmValue = rad2PwmMiddleMotor(m_leg.getMotorAngle(i,1));
    pwmValue = rad2PwmOuterMotor(m_leg.getMotorAngle(i,2));
  }
  if(m_pSharedData->lock())
  {
    m_pSharedData->m_Pwm1Output = m_PwmOutput[0][0];
    m_pSharedData->m_Pwm2Output = m_PwmOutput[0][1];
    m_pSharedData->m_Pwm3Output = m_PwmOutput[0][2];
    m_pSharedData->unlock();
  }
}


void MotorControlModule::UpdatePwm()
{
  int tempPwm1=0;
  int tempPwm2=0;
  int tempPwm3=0;

  if(m_pSharedData->lock())
  {
    tempPwm1 = m_pSharedData->m_Pwm1Output;
    tempPwm2 = m_pSharedData->m_Pwm1Output;
    tempPwm3 = m_pSharedData->m_Pwm1Output;
    m_pSharedData->unlock();
  }
  this->m_PWM1Output = tempPwm1;
  this->m_PWM2Output = tempPwm2;
  this->m_PWM3Output = tempPwm3;
}


void MotorControlModule::UpdateMotors()
{
  TIM4->CCR1 = this->m_PwmOutput[0][0]; //this->m_PWM1Output;//leg
      1 motor 1
  TIM4->CCR4 = this->m_PwmOutput[1][0]; //this->m_PWM1Output;//leg
      2 motor 1
  TIM3->CCR3 = this->m_PwmOutput[2][0]; //this->m_PWM1Output;//leg
      3 motor 1
  TIM2->CCR2 = this->m_PwmOutput[3][0]; //this->m_PWM1Output;//leg
      4 motor 1
```

```
    TIM5->CCR1 = this->m_PwmOutput[4][0]; //this->m_PWM1Output;//leg
        5 motor 1
    TIM5->CCR4 = this->m_PwmOutput[5][0]; //this->m_PWM1Output;//leg
        6 motor 1


    TIM4->CCR2 = this->m_PwmOutput[0][1]; //this->m_PWM2Output;//leg
        1 motor 2
    TIM3->CCR1 = this->m_PwmOutput[1][1]; //this->m_PWM2Output;//leg
        2 motor 2
    TIM3->CCR4 = this->m_PwmOutput[2][1]; //this->m_PWM2Output;//leg
        3 motor 2
    TIM2->CCR3 = this->m_PwmOutput[3][1]; //this->m_PWM2Output;//leg
        4 motor 2
    TIM5->CCR2 = this->m_PwmOutput[4][1]; //this->m_PWM2Output;//leg
        5 motor 2
    TIM9->CCR1 = this->m_PwmOutput[5][1]; //this->m_PWM2Output;//leg
        6 motor 2

    TIM4->CCR3 = this->m_PwmOutput[0][2]; //this->m_PWM3Output;//leg
        1 motor 3
    TIM3->CCR2 = this->m_PwmOutput[1][2]; //this->m_PWM3Output;//leg
        2 motor 3
    TIM2->CCR1 = this->m_PwmOutput[2][2]; //this->m_PWM3Output;//leg
        3 motor 3
    TIM2->CCR4 = this->m_PwmOutput[3][2]; //this->m_PWM3Output;//leg
        4 motor 3
    TIM5->CCR3 = this->m_PwmOutput[4][2]; //this->m_PWM3Output;//leg
        5 motor 3
    TIM9->CCR2 = this->m_PwmOutput[5][2]; //this->m_PWM3Output;//leg
        6 motor 3
}
```

C.7 Leg Control

C.7.1 LegControl.h

```
#ifndef LEGCONTROL_H_
#define LEGCONTROL_H_

#include <cmath>
#include "SharedDataModule.h"

const double PI = 3.141592;
#define NUMBER_OF_LEGS 6
#define MOTORS_PR_LEG 3
#define MAX_POSITION_POINTS 11
#define X_COORDINATE 0
#define Y_COORDINATE 1
#define Z_COORDINATE 2
const double SECS_PR_ENDPOINT = 0.75;
const double LEG_POSITION_LOWER_BOUND = -10;
const double LEG_POSITION_UPPER_BOUND = 5;

class LegControl
{
public:
  LegControl(SharedDataModule &SharedData);
  ~LegControl();

  void RunLegController();
  double getMotorAngle(int leg, int motornr);

private:
  SharedDataModule *m_pSharedData;
  int m_counter;
  int m_legPointCounter;
  double m_centerToLeg;
  double m_innerLegLength;
  double m_middleLegLength;
  double m_outerLegLength;
  double m_elevation;
  double **m_MotorAngles;
  double *m_LegAngles;
  double **m_LegPositionCentre;
  double **m_LegPosition;

  double m_X;
  double m_Y;
```

```
  double m_Z;

  void UpdateEndpoints(int i);          //this acts as the walking
      algorithm
  void CalculateMotorPositions(int i);//this acts as the leg
      controller
};
#endif /* LEGCONTROL_H_ */
```

C.7.2 LegControl.cpp

```
#include "LegControl.h"

LegControl::LegControl(SharedDataModule &SharedData)
:m_counter(0),
m_legPointCounter(0),
m_centerToLeg(8.5),
m_innerLegLength(3),
m_middleLegLength(10),
m_outerLegLength(15),
m_elevation(LEG_POSITION_UPPER_BOUND),
m_X(30),
m_Y(0),
m_Z(-5)
{
  m_pSharedData = &SharedData;
  // Initializes angles for the legs
  m_LegAngles = new double[NUMBER_OF_LEGS];
  m_LegPositionCentre = new double*[NUMBER_OF_LEGS];
  for (int i=0; i<NUMBER_OF_LEGS; i++)
  {
    m_LegAngles[i] = -i*2*PI/NUMBER_OF_LEGS;
    while(m_LegAngles[i] < 0)
    {
      m_LegAngles[i] += (2*PI);
    }
    m_LegPositionCentre[i] = new double[MOTORS_PR_LEG];
    m_LegPositionCentre[i][X_COORDINATE] = cos(this->m_LegAngles[
        i])*(this->m_centerToLeg + 10);
    m_LegPositionCentre[i][Y_COORDINATE] = sin(this->m_LegAngles[
        i])*(this->m_centerToLeg + 10);
    m_LegPositionCentre[i][Z_COORDINATE] = 0;
  }

  // Initializes the angle outputs for all the motors;
  m_MotorAngles = new double*[NUMBER_OF_LEGS];
  for(int i=0; i< NUMBER_OF_LEGS; i++)
  {
    m_MotorAngles[i] = new double[MOTORS_PR_LEG];
    for(int j=0; j<MOTORS_PR_LEG; j++)
    {
      m_MotorAngles[i][j] = 0;
    }
  }
```

```
  //Endpoint of leg movement:
  m_LegPosition = new double*[MAX_POSITION_POINTS];
  for(int i=0; i<MAX_POSITION_POINTS; i++)
  {
    m_LegPosition[i] = new double[3];
    for(int j=0; j<3; j++)
    {
      m_LegPosition[i][j] = 0;
    }
  }
  //COORDINATES FOR WALKING PATTERN HERE:
  m_LegPosition[0][X_COORDINATE] = 0;
  m_LegPosition[0][Y_COORDINATE] = 0;
  m_LegPosition[0][Z_COORDINATE] = 0;

  m_LegPosition[1][X_COORDINATE] = 0;
  m_LegPosition[1][Y_COORDINATE] = -5;
  m_LegPosition[1][Z_COORDINATE] = 0;

  m_LegPosition[2][X_COORDINATE] = 0;
  m_LegPosition[2][Y_COORDINATE] = 0;
  m_LegPosition[2][Z_COORDINATE] = 6;

  m_LegPosition[3][X_COORDINATE] = 0;
  m_LegPosition[3][Y_COORDINATE] = 5;
  m_LegPosition[3][Z_COORDINATE] = 0;

  m_LegPosition[4][X_COORDINATE] = 0;
  m_LegPosition[4][Y_COORDINATE] = 0;
  m_LegPosition[4][Z_COORDINATE] = 0;

  m_LegPosition[5][X_COORDINATE] = 0;
  m_LegPosition[5][Y_COORDINATE] = -5;
  m_LegPosition[5][Z_COORDINATE] = 0;
}

LegControl::~LegControl()
{


}

void LegControl::RunLegController()
{
  //Update Elevation
```

```
double tempElevation = this->m_elevation;
double elevationRate = 0;
if(m_pSharedData->lock())
{
  elevationRate = m_pSharedData->m_elevationRate;
  m_pSharedData->unlock();
}
tempElevation += (elevationRate/100);// gives a max elevation
    rate on 1 cm/second
if(tempElevation > LEG_POSITION_UPPER_BOUND)
{
  tempElevation = LEG_POSITION_UPPER_BOUND;
}
else if(tempElevation < LEG_POSITION_LOWER_BOUND)
{
  tempElevation = LEG_POSITION_LOWER_BOUND;
}
this->m_elevation = tempElevation;
if(m_pSharedData->lock())
{
  m_pSharedData->m_elevation = this->m_elevation;
  m_pSharedData->unlock();
}

this->m_counter++;
if(this->m_counter >= (SECS_PR_ENDPOINT*100))
{
  this->m_counter = 0;
  this->m_legPointCounter++;
  if(m_legPointCounter > 3)
    m_legPointCounter = 0;
}

if(m_pSharedData->lock())
{
  m_pSharedData->m_walkingCounter = (uint32_t)m_legPointCounter
      ;
  m_pSharedData->unlock();
}

for(int i=0; i < NUMBER_OF_LEGS; i++)
{
  this->UpdateEndpoints(i);
  this->CalculateMotorPositions(i);
}
```

```
}

void LegControl::UpdateEndpoints(int i) //This is the walking
    algorithm
{
  int pos = this->m_legPointCounter;
  if(i%2 == 0)
  {
    pos += 2;
  }
  double tempX1 = this->m_LegPosition[pos][X_COORDINATE];
  double tempX2 = this->m_LegPosition[pos + 1][X_COORDINATE];
  double deltaX = tempX2-tempX1;

  double tempY1 = this->m_LegPosition[pos][Y_COORDINATE];
  double tempY2 = this->m_LegPosition[pos + 1][Y_COORDINATE];
  double deltaY = tempY2-tempY1;

  double tempZ1 = this->m_LegPosition[pos][Z_COORDINATE];
  double tempZ2 = this->m_LegPosition[pos + 1][Z_COORDINATE];
  double deltaZ = tempZ2-tempZ1;

  double DiffX = tempX1 + deltaX*(double)m_counter/(
      SECS_PR_ENDPOINT*100);
  double DiffY = tempY1 + deltaY*(double)m_counter/(
      SECS_PR_ENDPOINT*100);
  double DiffZ = tempZ1 + deltaZ*(double)m_counter/(
      SECS_PR_ENDPOINT*100);

  double steeringAngle = 0;
  double speed = 0;
  if(m_pSharedData->lock())
  {
    steeringAngle = m_pSharedData->m_steeringAngle;
    speed = m_pSharedData->m_speed;
    m_pSharedData->unlock();
  }

  this->m_X = m_LegPositionCentre[i][X_COORDINATE] + speed*(DiffX
      *cos(steeringAngle) - DiffY*sin(steeringAngle));
  this->m_Y = m_LegPositionCentre[i][Y_COORDINATE] + speed*(DiffX
      *sin(steeringAngle) + DiffY*cos(steeringAngle));
  this->m_Z = m_LegPositionCentre[i][Z_COORDINATE] + speed*DiffZ
      + this->m_elevation;
```

```
  if(m_pSharedData->lock())
  {
    m_pSharedData->m_endX = m_X;
    m_pSharedData->m_endY = m_Y;
    m_pSharedData->m_endZ = m_Z;
    m_pSharedData->unlock();
  }
}

void LegControl::CalculateMotorPositions(int i)// This is the leg
     control.
{
  double legPosition[3];
  double tempAngle = m_LegAngles[i];
  legPosition[X_COORDINATE] = cos(this->m_LegAngles[i])*
      m_centerToLeg;
  legPosition[Y_COORDINATE] = sin(this->m_LegAngles[i])*
      m_centerToLeg;
  legPosition[Z_COORDINATE] = 0;
  double relativeDistance[3];
  relativeDistance[X_COORDINATE] = m_X - legPosition[X_COORDINATE
      ];
  relativeDistance[Y_COORDINATE] = m_Y - legPosition[Y_COORDINATE
      ];
  relativeDistance[Z_COORDINATE] = m_Z - legPosition[Z_COORDINATE
      ];

  double angle1 = atan2(relativeDistance[Y_COORDINATE],
      relativeDistance[X_COORDINATE]) - this->m_LegAngles[i];
  while(angle1 < -PI/2)
  {
    angle1 += 2*PI;
  }


  double dist_xy = sqrt(relativeDistance[X_COORDINATE]*
      relativeDistance[X_COORDINATE] + relativeDistance[
      Y_COORDINATE]*relativeDistance[Y_COORDINATE]);
  double dist_xyz = sqrt(dist_xy*dist_xy + relativeDistance[
      Z_COORDINATE]*relativeDistance[Z_COORDINATE]);

  double D = ((dist_xyz*dist_xyz - this->m_middleLegLength*this->
      m_middleLegLength - this->m_outerLegLength*this->
      m_outerLegLength)/(2*this->m_middleLegLength*this->
      m_outerLegLength));
```

```
  double angle3 = -acos(D);
  double angle21 = atan2(relativeDistance[Z_COORDINATE], dist_xy)
      ;
  double E = (this->m_middleLegLength*this->m_middleLegLength +
      dist_xyz*dist_xyz - this->m_outerLegLength*this->
      m_outerLegLength)/(2*this->m_middleLegLength*dist_xyz);
  double angle22 = acos(E);
  double angle2 = angle21 + angle22;

  this->m_MotorAngles[i][0] = angle1;
  this->m_MotorAngles[i][1] = angle2;
  this->m_MotorAngles[i][2] = angle3;
}

double LegControl::getMotorAngle(int leg, int motornr)
{
  double angle = this->m_MotorAngles[leg][motornr];
  return angle;
}
```

C.8 Arduino PWM reader

This is the only piece of code that is not written for the Discovery STM32F4. It is a piece of code that is written for an Arduino Nano, which measures the duty cycles of the PWM signals outputs from the radio remote controller's receiver, and sends the values via a serial connection.

```
double channel[4];

void setup() {
  pinMode(2, INPUT);
  pinMode(3, INPUT);
  pinMode(4, INPUT);
  pinMode(5, INPUT);
  Serial.begin(9600);
}

void loop() {
  for(int i=0; i<4; i++)
  {
    channel[i] = pulseIn(i+2, HIGH);
  }
  for(int i=0; i<4; i++)
  {
    Serial.print("Ch");
    Serial.print(i);
    Serial.print(":");
    Serial.print(channel[i]);
    Serial.print(",");
  }
  Serial.print('\n');
  delay(250);
}
```