UiA University of Agder

Bachelor

Flight Controller Design and Implementation for UiA Drone Project

Eirik Helle

Henrik Hansen Togba

Jakob Underdal Finnvold

Fridtjof Herberg Lunde

SUPERVISOR
Kristian Muri Knausgard

University of Agder, 2022
Faculty of Engineering and Science
Department of Engineering and Sciences

Acknowledgments

The bachelor thesis marks the end of the Mechatronics bachelor program at the University of Agder. With a bachelor's degree in Mechatronics, the path ahead is filled with possibilities, where some might feel the urgent call of the working life and others might feel intrigued to explore the field even further with a master's degree. The bachelor's program has been highly interesting and educational, with the introduction to everything from control systems to 3D modeling and computer languages.

This project introduces many of the fields of Mechatronics, with elements ranging from basic modeling of drone components to circuit board design and software interfaces. The project has been, like the rest of the bachelor program, very educational, fun, and especially time-consuming. Working with drones has also given us a different perspective on the complexity of such systems, and also the benefits the drones can have for the future of the world.

We want to give our greatest thank you to the faculty for giving us the opportunity to work on this platform, and for providing us with the help we needed. The service engineers of the mechatronics lab were more than eager to help us out with equipment and guidance. We also want to thank our excellent supervisor Kristian Muri Knausgård, for giving us great advice and a low threshold policy for asking questions. The communication has been impeccable.

Personally, the motivation in the group has been high. The group shares a similar passion for mechatronics, where some like some fields more than others. For this reason, it was very easy to assign the different tasks to the ones that wanted them, which created a good synergy within the group. Though the group members had different tasks, it was never an issue helping each other, and the project contributed to a better development within the field. The learning curve has been steep, but the outcome has given us many valuable lessons and even more knowledge.

Grimstad, 20. Mai 2022

Henrik H. Togba

Eirik Helle

Fridtjof H. Lunde

Jakob U. Finnvold

Abstract

UiA's drone project has been given as a bachelor assignment since 2017. Where the end goal is to have a functional UiA-made drone to be implemented in a swarm. The assignment for this year's project will take a closer look at the design of a new flight controller for these drones, and test a new framework. The project is based on last year's bachelor thesis, where the flight controller used was a commercially produced controller that made the drone exceed the weight requirement [79].

The flight controller will be a microprocessor in combination with a diversity of external sensors mounted on or in connection with a self-developed printed circuit board (PCB). It is also a goal to test NASA's F' framework, evaluate it for further use, and decide whether to stick with it or adopt another framework. Further in the report, it is described why F' was not the best fit for the drone at hand. The new flight controller was also evaluated, where the protocol for data transfer was especially looked into. The connector type HIROSE between the carrier board and the Raspberry Pi Compute Module 4 was also evaluated, and was found to create a lot of problems due to the small size and sensitivity on placement. These connector problems were then traced to problems with sensor readings on the flight controller. Conclusions were drawn to recommend further development of the software for the flight controller and to look into better options for debugging the HIROSE connector and data transfer tracks.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background and motivation

The UiA drone project has been under development for a few years with the goal of developing a platform for research and educational purposes. The previous projects have revolved around designing a modular quadcopter, and implementing said drones in swarm applications. The drones have previously been designed to work with Qualysis Motion Capture, and also to be stacked on top of one another on a launcher [72][79].

The previous iteration of the project focused on, as mentioned, developing a launcher for the drones. This demanded a lot of time, and the solution was to use a commercially available flight controller to make the testing process easier. This resulted in a heavy drone, which did not meet the weight requirements of 250 grams. Referring to §51 of "*Forskrift om luftfartøy som ikke har fører om bord*": *"Aircrafts which have a MTOM of 250 grams or less, can be flown VLOS, EVLOS or BLOS, but not higher than 50 meters above the ground or water"* [77]. This requirement is therefore a must if the UiA drones are to be operated outside licensed bounds, and that is why this iteration of the project focuses on meeting this requirement. These rules do not apply indoors, but as the project is supposed to develop into a swarm, it is necessary to be able to fly it outside.

The focus is still to preserve a highly modular and low-cost drone. To form a swarm of expensive drones would be good to avoid due to the fact that a swarm can contain over 100 drones. Making a drone with as few, but still good quality components as possible, would make the assembly of the drones easy, intuitive, and prone to mass production.

1.2 State of the art

The current technology for swarm drones or drones in itself has come quite far in the last few years and the use of drones in industry, defense systems, or personal use has increased rapidly just since 2019 [56]. The rapid increase in drones also means that there is a lot more focus and research on the technology concerning flight controllers. There is a great number of manufacturers and suppliers on the current market. For example, the Pixhawk 4 which was used in the previous bachelor thesis [79], is rated number 3 out of the best flight controllers in 2022 by Staaker.com [117]. DJI Naza-M V2 is on second place and Hobbypower KK2 in first place on the list. There are some differences between them, like the self-update feature on the firmware, that is available on Hobbypower KK2 and some sensor differences with the IMU innovative calibration available on DJI Naza-M V2. There are also some complete drones with possibilities to be implemented in a swarm like Crazyflie 2.1 [21] [2], which will fit in the palm of your hand and only weighs 27g. Crazyflie 2.1 is in addition to being very small, an open-source platform that allows the consumers to use it as they desire. It also uses BMI088 and BMP388 as sensors, which is explained in section 4.4.4.

1.3 Task description

The task is to design and implement a new flight controller for the UiA drones, and by that liquidate the Pixhawk 4 Mini setup on the existing drones. This involves designing a new PCB with the necessary sensors, and using the Raspberry Pi Compute Module 4 [100] accompanied by a microcontroller to finalize a more compact, lightweight solution to meet the weight requirements of the drones. This also introduces the need for a real-time operating system and a platform to make the necessary flight algorithms. The framework F' by NASA is to be tested and evaluated with the flight controller to find out if it is a suitable fit for the drones together with a real-time Linux patch as the real-time operating system. The algorithms are to be developed in C/C++. Additional evaluation of a Time of Flight sensor is also to be conducted and implemented into the drone shell.

Chapter 2

Theory

2.1 Electronics

2.1.1 Sensors

Inertial Measurement Unit

An Inertial Measurement Unit, or IMU, is a sensor measuring inertial force and angular rate. The IMU comes in various types and mainly consists of an accelerometer and gyroscope. Many IMUs also include a magnetometer. They commonly exist from ranges of 3 DOF to 9 DOF. DOF stands for degrees of freedom, which comes from the motion along and about each axis of the Cartesian coordinate system. For drone applications, an IMU would provide relevant data concerning the pitch, roll, and yaw of the drone.

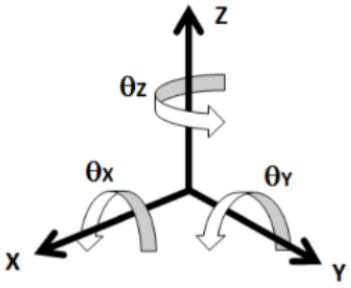


Figure 2.1: 6 Degree of Freedom Representation [52]

Accelerometer

The accelerometer measures inertial force based on Newton's 2nd law. By rearranging the equation, the acceleration along the Cartesian axis can be calculated as shown in equation 2.1.

$$F = ma \longrightarrow a = \frac{F}{m} \tag{2.1}$$

An accelerometer measures specific force, which is the time rate of change of velocity, relative to local gravitational space. This means that it treats the accelerometer, or what the accelerometer is mounted to, as a free body, measuring the movement it has, relative to itself. It has to measure relative to itself due to the constant static acceleration of $9.81 m/s^2$ on earth, which would affect its measurements if it was relative to the earth. The accelerometer senses movement of a proof mass from a nominal position inside the case of the accelerometer, and from there calculates the specific force that has acted upon the case.[112]

If the accelerometer is placed at the center of the body, the angular velocity can also be calculated. However, this has low accuracy due to its fixed position. This does not mean that the

data is irrelevant, as it can be combined with a gyroscope to make the angular measurement even more accurate.

Gyroscope

The gyroscope is responsible for measuring the rotational motion of the axes of the coordinate system. It is useful for determining the orientation of the body in motion, as it measures the angular velocity in each direction.

The gyroscope measures angular velocity by the use of the Coriolis force [98]. From equation 2.2, the force is calculated based on the mass, angular velocity, and relative motion. The gyroscope then calculates the angular velocity (ω) based on the Coriolis force applied to the body. The sensor contains vibrating elements, and when the Coriolis force makes these elements vibrate, the drive arms in the sensor move in different directions depending on the force. The sensing arms in the sensor are affected by the motion of the drive arms, and depending on how the drive arms move, the sensing arms provide an electrically charged output. The change in this output is used to measure the rotational force and with this, the angular velocity is calculated [42].

$$F = 2m\omega V \tag{2.2}$$

$$\omega_{xyz} = \frac{d\theta_{xyz}}{dt} \tag{2.3}$$

When the angular velocity is read from the sensor, the signal is integrated. This gives a very precise output, but with the downside of only working for a short period of time. If the system is stable in the 3D space, the angular velocity is calculated as the same every time. When this is integrated, it will produce an offset due to the same value being integrated over and over. Using a highpass filter on the gyroscope is therefore necessary to avoid this scenario. The gyroscope provides an additional 3 degrees of freedom, which when combined with the accelerometer provides an IMU with 6-DOF.

Magnetometer

In addition, an IMU can have a third sensor, the magnetometer. The magnetometer measures, put simply, the earth's magnetic field and fluctuations within it. It works in essence like a compass, as it can determine the orientation of the body based on the earth's poles [59]. This is because it detects the magnetic flux density of the body's location, and based on this can determine where the earth's poles are relative to the body's placement in the air [71]. This provides an additional 3 degrees of freedom, which implies that an IMU containing all these sensors would have 9 degrees of freedom. Based on the application, both 6 and 9 DOF IMUs would be good candidates.

Complementary filter

When using an IMU, the accelerometer and gyroscope give different values. In a drone context, the useful information from these sensors is pitch, roll, and yaw. As the accelerometer provides acceleration along the axes, this can be converted to pitch and roll by using these equations 2.5 2.6. These values are overall good, but can be severely manipulated by noise and in addition, the yaw element is not well represented by the accelerometer as it is based on gravity. The gyroscope gives us the angular velocity, which essentially is the pitch and roll. The magnetometer provides us with the orientation of the body, which will be the yaw based on equation 2.7.
By using a complementary filter, these values can be combined to produce an accurate representation of the pitch and roll. This is called sensor fusion. The yaw is generally only calculated based on the magnetometer, so this is not included in the filter. The accelerometer gives overall good values in the long run, while the gyroscope gives accurate short-term measurements. Fusing these readings together with equation 2.4 gives an accurate output of the variables. This equation favors the gyroscope data by 98% but the accelerometer data is also

included by 2% [103]. This makes the gyroscope values not drift, while still including the accelerometer values that hace long-winded decent readings. The constant can be changed if more or less of the accelerometer or gyroscope readings is wanted.

$$\theta = 0.98 * (\theta + gyrdata * dt) + 0.02 * (acceldata) \tag{2.4}$$

Equation 2.4: Complementary filter

Barometric Pressure Sensor

The Barometric Pressure Sensor is used to measure the current altitude the drone is located at. The sensor works by measuring the atmospheric pressure in its surroundings. The higher the drone is over the ocean level the lower the atmospheric pressure gets. The normal atmospheric pressure at sea level in Norway is 1013 (hPa), [29]. By using this method the drone will know the altitude its currently at.

A Piezo-resistive sensor, [120], which are made by using resistors placed on a silicon diaphragm for the purpose of reading the result of strain or physical force experienced on them, and it will give out different voltages depending on height which can be converted into altitude.

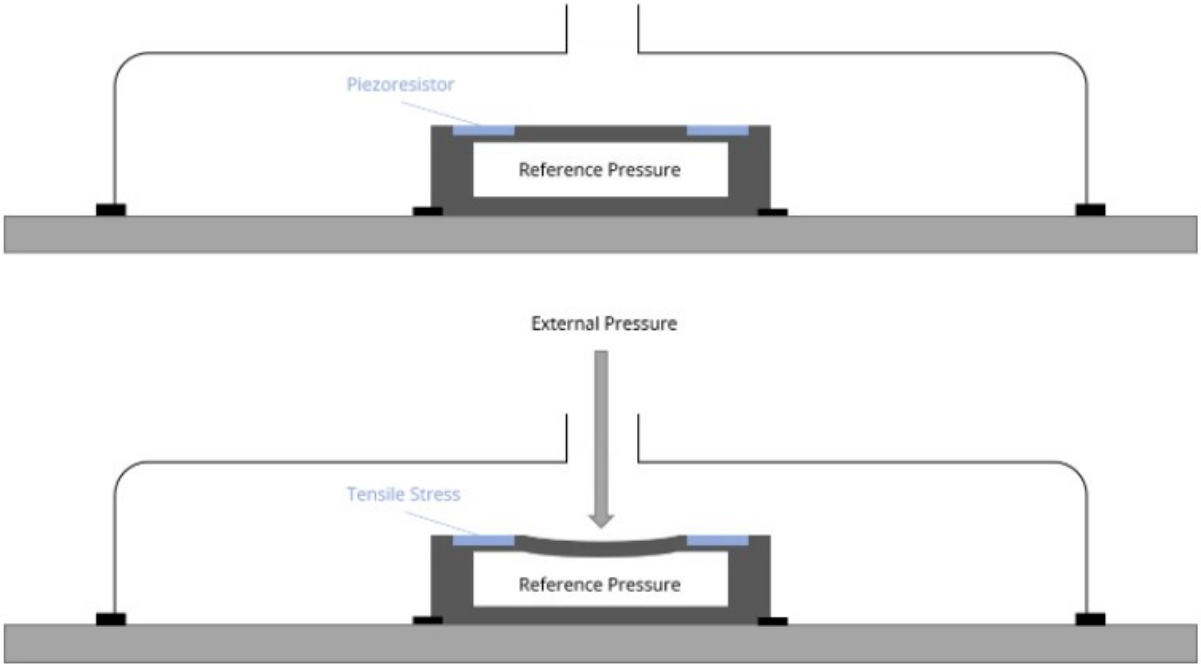


Figure 2.2: Example of how the barometric pressure sensor handles pressure [120]

Time of Flight Sensor

The Time of Flight (ToF) sensor is used for reading distances to objects within a field of view. The resolution of this reading is often 4x4 or 8x8 pixels. It can be described as a depth camera. A ToF sensor is able to work within an angle of view, which varieties by each model. The sensor is able to read distance by having an emitter that sends out a photon, that photon bounces off the target and gets read by an adjacent sensor. By multiplying the time between emitting and receiving to the constant light speed, C, and dividing this by two, the distance to the object/target is revealed. It is necessary to divide by two because the photon has traveled twice the distance.

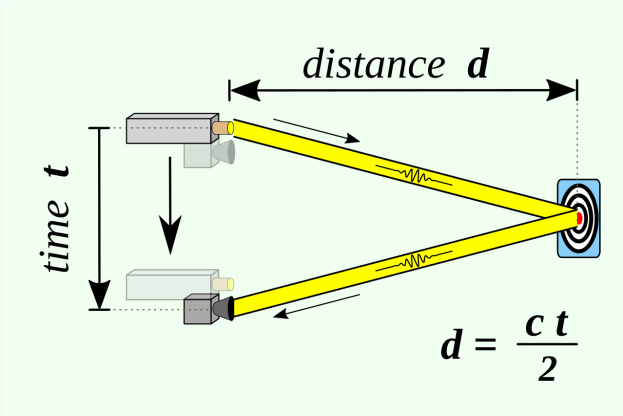


Figure 2.3: Visual representation of how a ToF sensor operates, [134].

2.1.2 Carrier board

A carrier board, also known as a Printed Circuit Board (PCB), is an extension to the main board of the computer unit. For example, the Raspberry Pi CM4 [100]. Carrier boards can be either bought premade or designed from scratch to fit the needs of the project. In the case of self-design, the board schematics would be sent to a manufacturer for production. With regards to the budget, it might be wiser to go for something that is already produced, but as other factors enter the list of needs, this might change. For instance, it might be needed to cut down on the space or the weight of the board, and therefore it will be necessary to design a PCB.

2.1.3 Low dropout voltage regulator - LDO

A low dropout regulator is a DC linear voltage regulator that can regulate the voltage even if the input is very close to the output. The benefit of an LDO is that it does not produce any switching noise on the signal. This is good for sensitive sensors. The downside of LDOs is that they in some way must dissipate the power that is left, and since no switching occurs, the additional energy left from the regulating becomes heat.[108]

2.1.4 Decoupling capacitors

There has to be capacitors fitted close to the components on power supplying lines. These are used to ensure that a drop in voltage does not affect the component. They work as energy storage close to the component and will take away energy if the voltage gets too high, and the other way around they will supply extra energy if the voltage drops too low. This stabilizes the voltage on the power supply for the component [32]. An example can be seen underneath in figure 2.4 which shows capacitors reducing noise in the circuit. Decoupling capacitors also have to be close to the circuit they are decoupling.

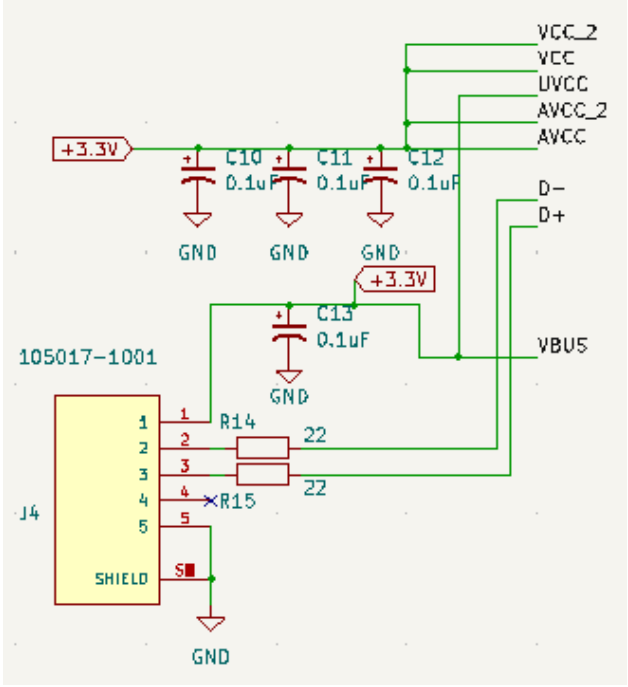


Figure 2.4: How decoupling capacitors (C10, C11, and C12) are used for the power supply track to the ATmega32U4 (Section 4.4.2) following the manufacturer's recommendations.

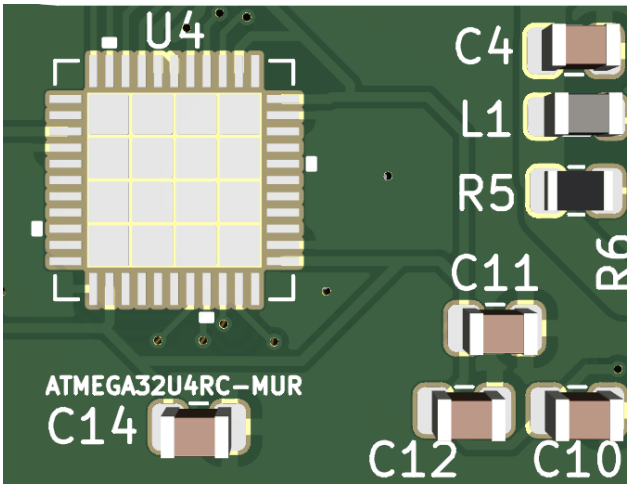


Figure 2.5: 3D model of PCB with decoupling capacitors (C10, C11, and C12) located close to the ATmega32U4

2.2 Mechanics

2.2.1 Drone dynamics

A drone has three directions of movement. These are referred to as the pitch, roll, and yaw. As shown in figure 2.6, the pitch is the rotation of the drone about the x-axis, the roll is the rotation about the y-axis and yaw is the rotation about the z-axis. The pitch can be referred to as forward and backward motion, the roll is the left or right motion and yaw is the center rotation.

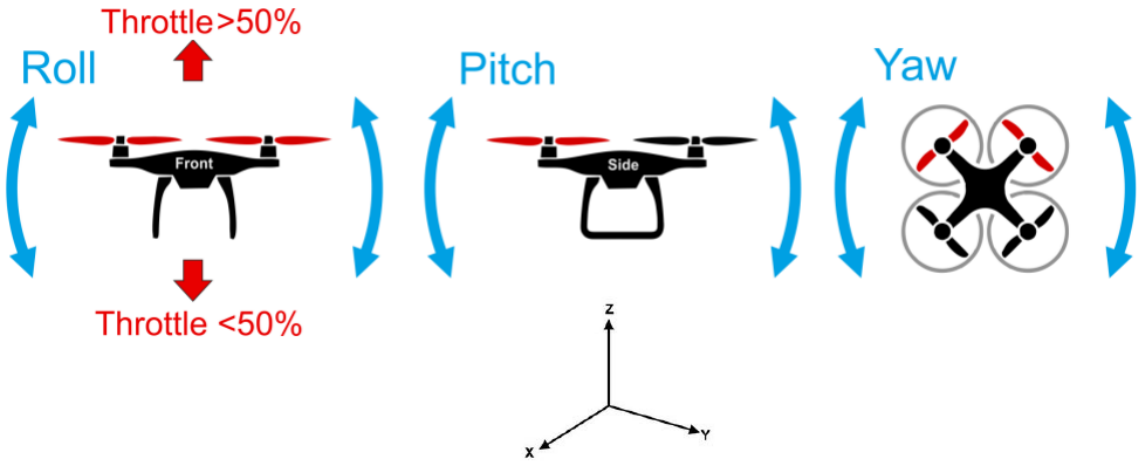


Figure 2.6: Basic movements of a quadcopter [104]

The pitch, roll, and yaw is calculated based on the equations below. The measurements from the onboard gyroscope give angular velocity outputs proportional to the pitch and roll, so these need no calculation.

$$Pitch_a = (180/\pi) * atan2(x, \sqrt{Ay^2 + Az^2}) \tag{2.5}$$

Equation 2.5: Calculating pitch from accelerometer

$$Roll_a = atan2(Ay, Az) \tag{2.6}$$

Equation 2.6: Calculating roll from accelerometer

$$Yaw_m = atan2(\frac{m_x}{m_y}) \tag{2.7}$$

Equation 2.7: Calculating yaw from the magnetometer

2.2.2 Acrylic glass - PMMA

Acrylic in the form of Polymethylmethacrylate (PMMA) is a transparent plastic material often used as a substitute for glass. This is because acrylic glass has mechanical properties that make it far more versatile than regular glass. Such as significantly better flexibility, ductility, and moldability [74]. This makes it ideal for the application where it is prone to impact.

2.2.3 PID controllers

A proportional integral derivative controller is broadly used in many control systems applications. The basics of a PID controller is that it calculates errors based on a given setpoint provided by the designer. This could be the desired level in a water tank, where the PID controller can release and take in the fluid to make the water in the tank stay leveled. Another example is the stabilization of a drone. When flying a drone, it is desired to have a stable platform to fly on. With a PID controller, the drone can stay stable by reacting to the pilot's commands and making sure there is no overshoot to where the drone could spiral out of control [111].

The PID controller consists of three parts. One proportional part, one integral part, and one derivative part. These parts form a general equation for the output error, which looks like the one represented in equation 2.8.

$$u(t) = K_p e(t) + K_i \int_0^t e(t)dt + K_d \frac{de(t)}{dt} \tag{2.8}$$

There are three constants in each part of the equation. These are to be tuned in each implementation of a PID controller, to achieve the appropriate output based on the application. The first constant is the proportional constant K_p. This is multiplied with the error, which is *e(t)*, ergo its name, proportional. This is most commonly the largest constant, as it is directly proportional to the error, and to quickly compensate for an error, this constant is a vital one. The next constant is the integral constant K_i. This is multiplied by the error integrated over time. The integral accounts for the previous errors and uses this information to try to eliminate them. The last one is K_d, which is the derivative. It seeks to minimize the overshoot by slowing the correction factor that has been applied as the targeted setpoint is being approached [95]. The effect these constants have on the system is shown in table 2.1.

Table 2.1: Consequences of increasing the PID constants

	Rise time	Overshoot	Steady state error	Settling time
Kp	Decreases	Increases	Decreases	Decreases
Ki	Decreases	Increases	Eliminated	Increases
Kd	Decreases	Decreases	Not affected	Decreases

2.3 Software

2.3.1 KiCad - PCB Design software

There are multiple choices in design software for PCBs, one of them is called Kicad, which is an all-in-one design software that lets you create your own circuit boards. This is done by firstly, drawing the required schematics for the board. Then a footprint has to be added to each component so that it knows what component goes onto the PCB. The software has a large variety of footprints already installed, but if a component is not frequently used, it might be needed to be downloaded as a library and added to the program. This was the case for the sensors and the ATmega microprocessor. Then a PCB can be created from the schematic, and components need to be placed on the board.

It is smart to locate components that are connected to each other close by one another, as this makes the routing of the tracks easier. Once all components have been placed, tracks that connect the matching pins together have to be routed. Holes in the board are called vias and are used to route a track past another one on the same layer, by going to the backside layer to go underneath the blockage, and then back to the front layer again. When all tracks are routed to the appropriate pins, the board is essentially complete. A DRC (Design Rule Check) is then performed to check for errors on the board, such as components too close to each other, or space between tracks being too small. On the other hand, the DRC will not detect errors in the schematic which then have been transferred to the board, such as wires that have been connected to the wrong places. Once the DRC is completed the software lets you create the files needed to send the design to production. [67]

2.3.2 Communication protocols

Inter-Integrated Circuit - I2C

Inter-Integrated Circuit, also known as I2C is a synchronous multi-controller/multi-target serial communication bus invented by Phillips in 1982. It is widely used for attaching lower-speed peripheral integrated circuits to processors and microcontrollers [132]. I2C is now supported by a large number of competitors to the creator and is now used widely in computing and electronics. I2C is very convenient given it only needs two wires between components. The first one is SDA - Serial Data, which sends data between controller and target. The second one is SCL - Serial Clock, which is the clock signal [94].

The benefit of using I2C is that it can be connected to multiple targets or multiple controllers on one I2C line. I2C does not have any form of a selection of slaves, therefore it needs to tell the slave whether it wants to send or receive data through the Serial Data line, as this is the only line that can send data. It does this by using a read/write bit at the beginning of the address that is sent. This lets the slave act appropriately. Since there can be numerous slaves connected on the same SDA and SCL lines to the master, there must also be a way to pick the slave the master wants to communicate with. This is done by using addresses. In an I2C address there is therefore first a read/write bit, and then the address of the wanted target. Every device on the line then compares the address sent to their own to see if they are the target. If it is a match the slave pulls the SDA line low for one bit to send back a signal telling it has been initiated [17]. See the visual representation in figure 2.7 below.

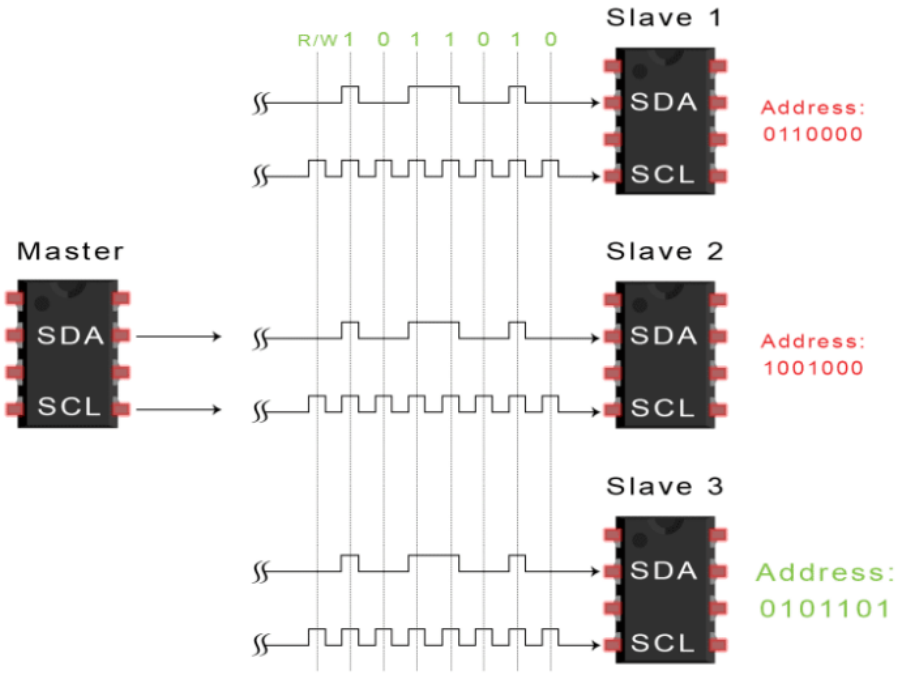


Figure 2.7: Example of I2C master and 3 slaves with different addresses from [17]

Serial Peripheral Interface - SPI

SPI is a synchronous serial communication interface used for short-distance communication, primarily in embedded systems and solutions. It was developed by Motorola in the mid-1980s and has since become a branch standard [133]. It works by using four wires between the two communication components. SPI is a slave-master type interface, where one component takes the master role, and one takes the slave role. Two of the wires between the components are therefore named MOSI for Master Input Slave Output, and MISO for Master Input Slave Output. The third cable is the SCLK which stands for Serial Clock and is controlled by the master. The fourth and last cable sometimes has different abbreviations, but usually, SS or CS, which stands for Slave Select or Chip Select. This signal is also controlled by the master. It is usually active low and indicates when data is being sent from the master [18]. An example of setup between two components is in the figure 2.8 below.

The benefit of SPI is that it has a fast transfer speed, up to 10 mbit/s, while I2C for example only can transfer at a max speed of 5 mbit/s [121]. It is also possible to connect multiple slaves to one master. MISO, MOSI, and SCLK lines can be used by multiple devices, but each device needs an independent SS. To select a chosen slave, the master pulls the SS of the chosen slave low. This is the reason SPI is a strong protocol in embedded systems. This feature makes SPI able to request information from a slave when it is needed, without the delay between transfers in a protocol that constantly emits signals. A visual representation of the SPI connection between one master and multiple slaves can be seen in figure 2.9 underneath. [37]

Figure 2.8: Example of Single Master to Single Slave SPI BUS from [133]

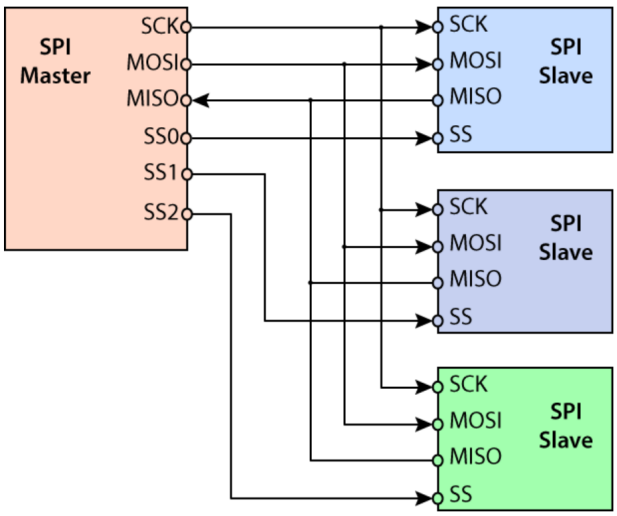


Figure 2.9: Example of Single Master to Multiple Slaves SPI BUS from [37]

2.3.3 NAT Protocol

The network address translation protocol is a protocol used to map and share private IP addresses of devices to public addresses on the same network [116]. This protocol is used in for example routers, where every connected device will receive an IP address provided by the protocol. The device will then have an internal address and a public address. There are three types of this protocol, which are the static and dynamic NAT along with PAT which stands for "port address translation" [34]. The dynamic NAT is one of the most used types. For example, a home router will have a range of public addresses, where it chooses one to give to the device. This can result in a different IP address each time the device is connected to the network.

The static NAT type assigns the same address to a device every time it is connected to the network [34]. For embedded systems that include for example a microprocessor with WiFi, this is a good way to use NAT. The device will then have a predictable address on the network, and thus the access through for example Secure Shell will be easier.

The PAT type allows for multiple devices to share a single public address [122]. It is similar to the dynamic NAT, although the dynamic NAT does not allow multiple devices on one address.

2.3.4 TCP Protocol

The Transmission Control Protocol (TCP) is a transportation protocol often used in addition to IP to ensure reliable transmission of packets in a packet-based system. TCP contains numerous mechanisms to solve problems that occur during packet-based messaging, like loss of packets, out-of-order, duplicate, and corrupt packets. A packet sent using TCP is formatted as a TCP segment, with each segment containing a header and data. The header can range in size between 20 and 60 bytes. The header contains data about the transfer, such as source port number, destination port number, and checksum [1].

The TCP connection between two computers is established using a "three-way handshake", where the first computer sends a packet with SYN(Synchronize) bit set to 1. The other computer sends back a packet with SYN bit set to 1 and ACK(Acknowledge) bit set to 1. The first computer then again sends back a packet with ACK bit set to 1, and the connection is established. When transmitting, the first computer sends a packet with data, and the receiving computer sends back a packet with ACK set to 1, and increases the acknowledgment number by the length of the received data[1].

2.3.5 Operating systems

Generally speaking, an operating system is a computer interface that allows the user to communicate with the system hardware in a civilized and user-friendly manner. The operating system is capable of performing tasks like file and memory management, as well as managing the input and output signals of the programs running on the computer [129]. A general way of describing the operating systems is that an operating system provides a suitable environment for the computer programs to execute on, as well as providing the user with an environment that is easy to use and simple to execute the programs on.

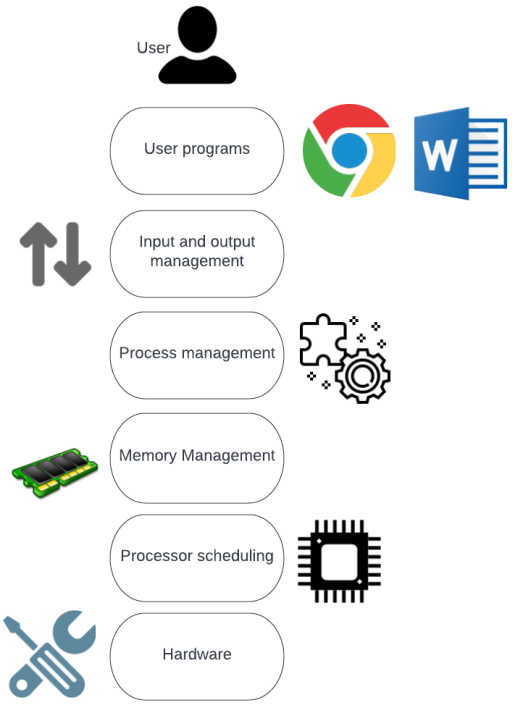


Figure 2.10: Layering of the Operating System

Most commonly, when asked about operating systems, people tend to mention the big brands like Windows, Linux, and Mac. These are all basic general-purpose operating systems, but these are not all the types of operating systems out there. One which is of great importance in this thesis is the real-time operating system.

GPOS

Figure 2.11: GPOS distributions [90]

As explained above, an operating system is a computer interface that allows for communication between users and hardware. The general-purpose operating system is just that. It is, simply told, an operating system for doing general things. Windows OS is one of these GPOSs, and it does what you expect. It provides a general-purpose user interface, which allows for a convenient and intuitive setup for the user. The ability to go on the internet and browse for files and videos, and also for playing video games. It does it all with good efficiency, and therefore a generally good system for the common public. Although these systems are always in development and updates come all the time, the general-purpose systems prioritize tasks more fairly [138]. This means that its ability to hit deadlines is not always good enough. For applications where this is highly necessary, like drone flight, the general-purpose systems are usually not good enough. This introduces the need for a real-time system that favors priority over fairness.

RTOS

The main difference between a GPOS and an RTOS is, obviously, the real-time capabilities. The real-time operating system operates based on priority threads set by the user to ensure that deadlines are met every time they are supposed to without unnecessary delay. This is essential when working with embedded systems which need these deadlines to be met. The RTOS provides support for scheduling actions, which make the system predictable and robust in embedded settings [118]. An excellent example is the drones in this project. If the deadlines of for example the sensor readings are not met and the latency is high, this can result in a slower conversion of the data, which will produce an incorrect PWM output from the flight controller. The motors then receive the wrong speed and the drone will either spiral out of control or shut down.

2.3.6 Frameworks

A framework in a programming context can be described as a set of tools which make the programming process faster [106]. The framework can contain pre-designed libraries, building tools, and simulation softwares. This can be highly useful for a developer if the coding part is complicated, as the framework can help implementing the code into the system.

Some examples of a framework is the Robotic Operating System, NET from Microsoft and F' by NASA. ROS provides a framework for robotic and embedded systems development [109], whereas NET is a framework used for website and desktop app design [82].

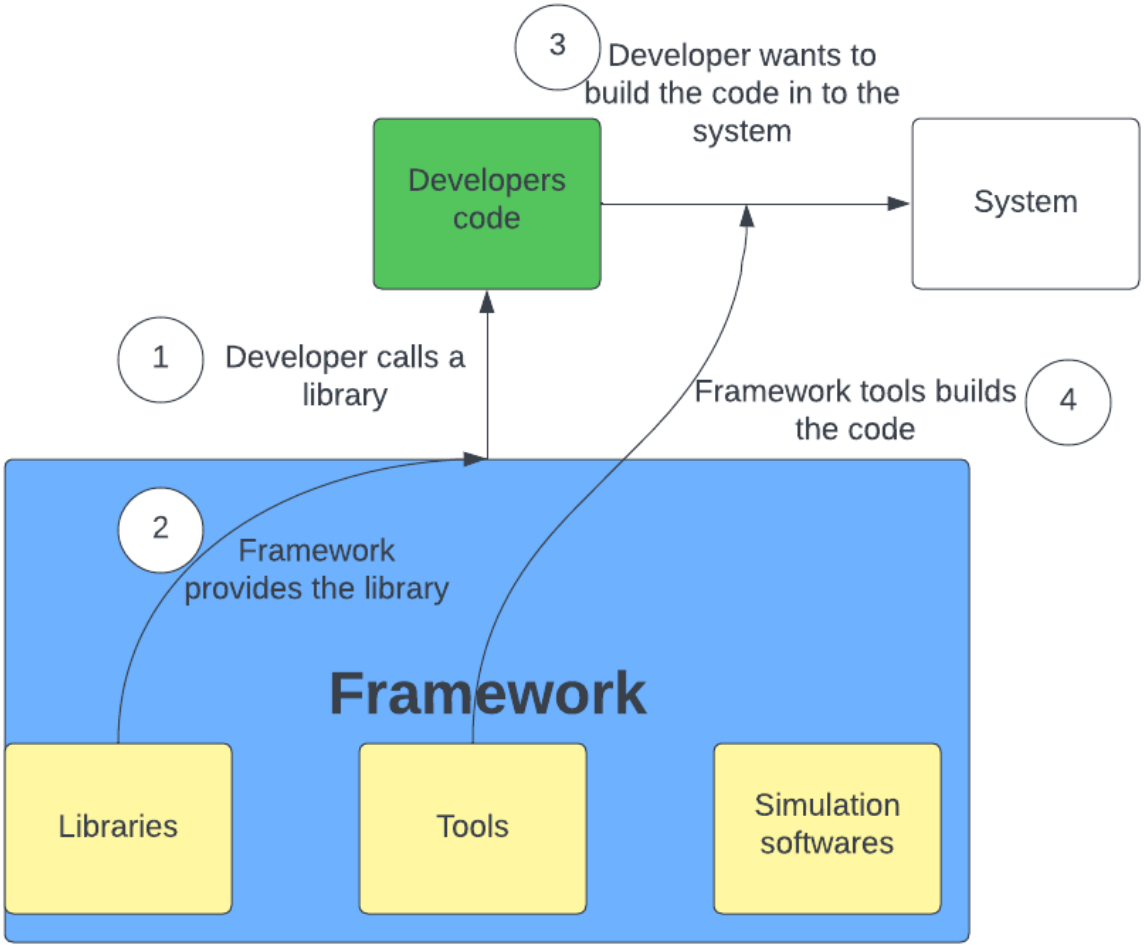


Figure 2.12: Basic framework usage

Chapter 3

Concept

The concept in this particular instance is derived by brainstorming the different aspects of the task and making a concept based on the ideas from the brainstorming. This was done in the initial startup of the project (Section 4.1).

3.1 Defining the development process

When first starting the project, the main task description provided in the bachelor's catalog was to evaluate the new NASA F' framework for the UiA drone project. This is quite a narrow description, and to include more aspects of mechatronics, the group decided to create a broader task description for the project together with the supervisor.

The previous iteration of the drone utilized a commercially developed flight controller to operate the drone. This led to the drone weighing 320 grams, and for this reason, the drone struggled to fly. Other previous iterations of the project had utilized microprocessors together with microcontrollers to control the drone. This process includes many aspects of mechatronics, including modeling and tweaking the drone shell, using sensors in embedded conditions, and using programming languages to read sensor data and operate the drone. It was therefore decided, in a group context, to design a new flight controller for the drones and to evaluate the NASA F' framework from the original task description.

3.2 Basics of a flight controller

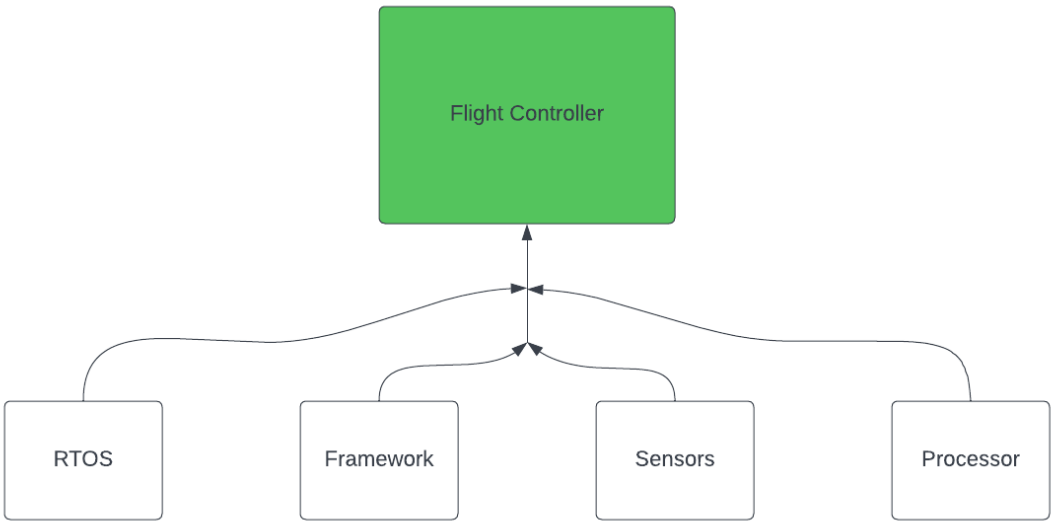


Figure 3.1: Flight Controller criteria

3.2.1 Flight controller sensors

For a flight controller to be able to control a drone, it needs a certain set of sensors and abilities to react properly. Firstly the controller needs the right type of sensors to receive data that can be acted upon to stabilize and fly a drone. This would be an Inertial Measurement Unit (IMU) and a barometer. These would measure the pressure and rotation of the drone, and send this data to the processor, which then reacts by controlling the motors to stabilize or fly the drone. Time of Flight (ToF) sensors would also be a great addition, as it opens up the possibility for anti-collision and distance measurement solutions. Based on this, it was decided to implement these sensors into the new flight controller.

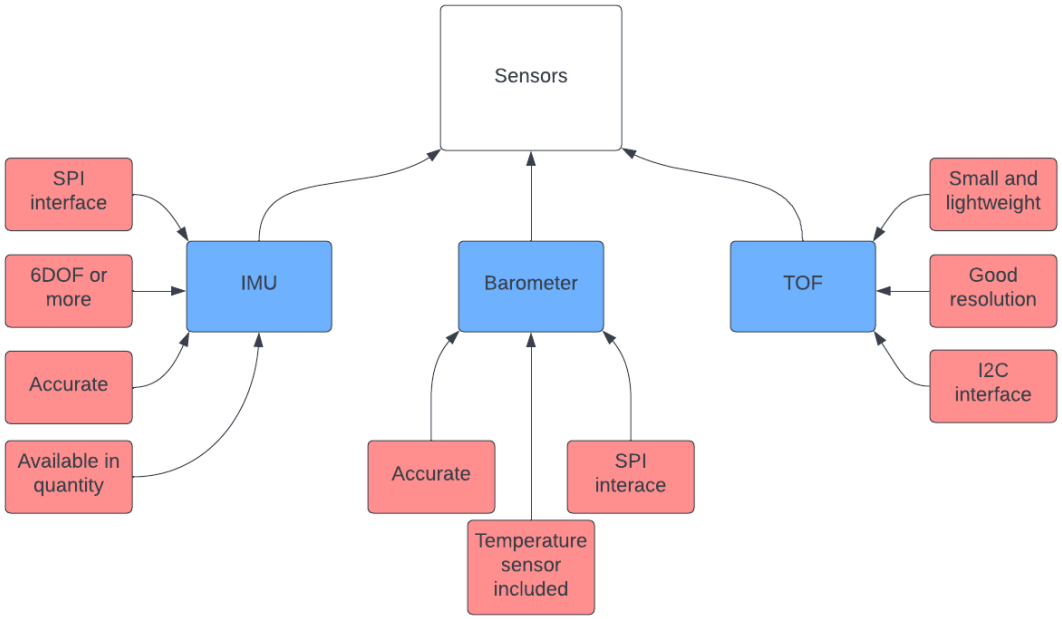


Figure 3.2: Sensor criteria

3.2.2 Operating systems

The controller would also have criteria for how fast it has to operate the drone, and how fast it has to react to changes from the sensors. Microscopic changes in the sensor readings must be acted upon if the drone is to be stable. Therefore it is necessary to run a Real-Time Operating System (RTOS) on the controller, to be able to react fast enough and hit deadlines on processing time. The use of RTOS instead of a normal operating system is to lower the latency on each process thread. Also, to make sure that no processes are put on a lower priority than they should, or interfere with other processes. There are many ROTS distributions. The most developed and used systems for drones are ChibiOS and NuttX, which are used in for example Pixhawks and the ArduPilot software[137]. For Linux distributions, the real-time patch for the Linux kernel is a good candidate.

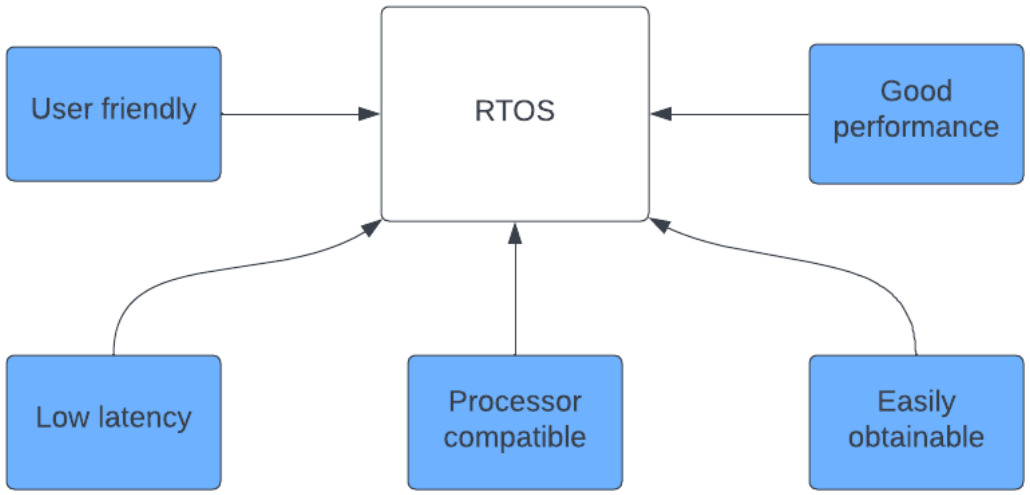


Figure 3.3: Real-Time Operating System criteria

3.2.3 Frameworks

For a drone, a framework could be a programming environment with appropriate libraries and tools to make the programming development phase easier and reduce the need to "bare-metal" code the algorithms. Most well-developed frameworks come with a predesigned ground user interface, which makes it easier to monitor and simulate drone operations and also a way to control the drone and map the drone path when flying. Such frameworks are for example the robotic operating system (ROS) or FPrime (F') which is a framework used for space-flight. It was therefore decided to evaluate the F' framework for the UiA drones, to see if it is a good candidate compared to ROS used in previous iterations of the project.

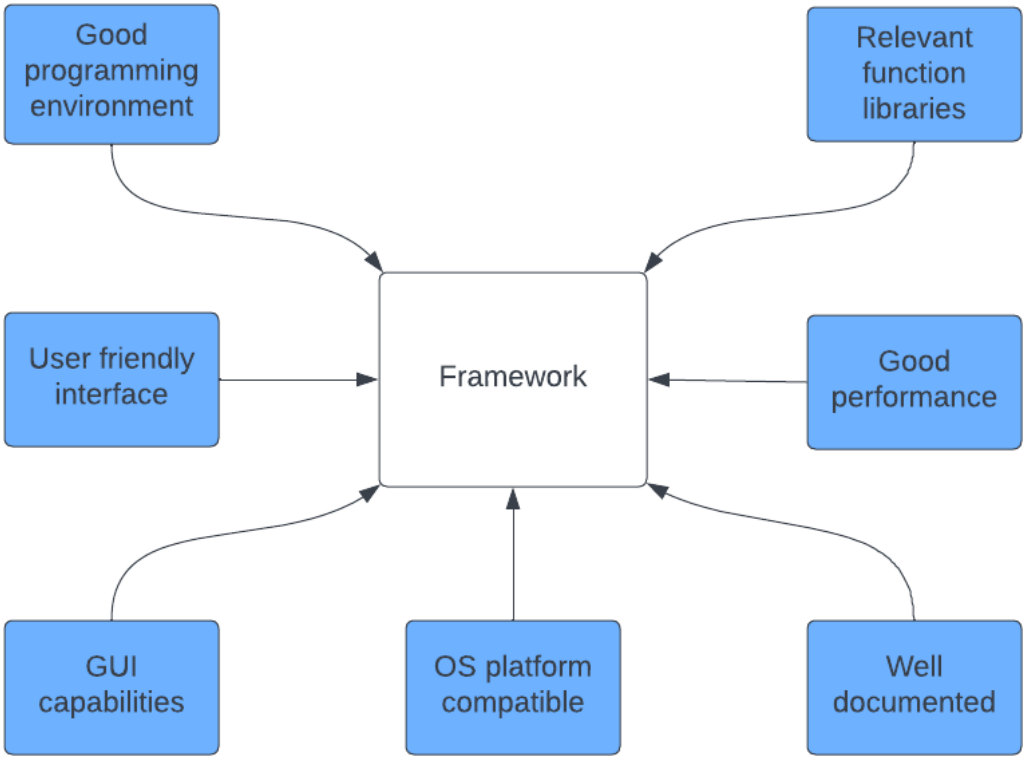


Figure 3.4: Framework criteria

3.3 Processor

A flight controller needs a processing unit to function optimally. It is important that this processor is fast, lightweight, and supports the interfaces of the sensors. It must also be compatible with the operating system.

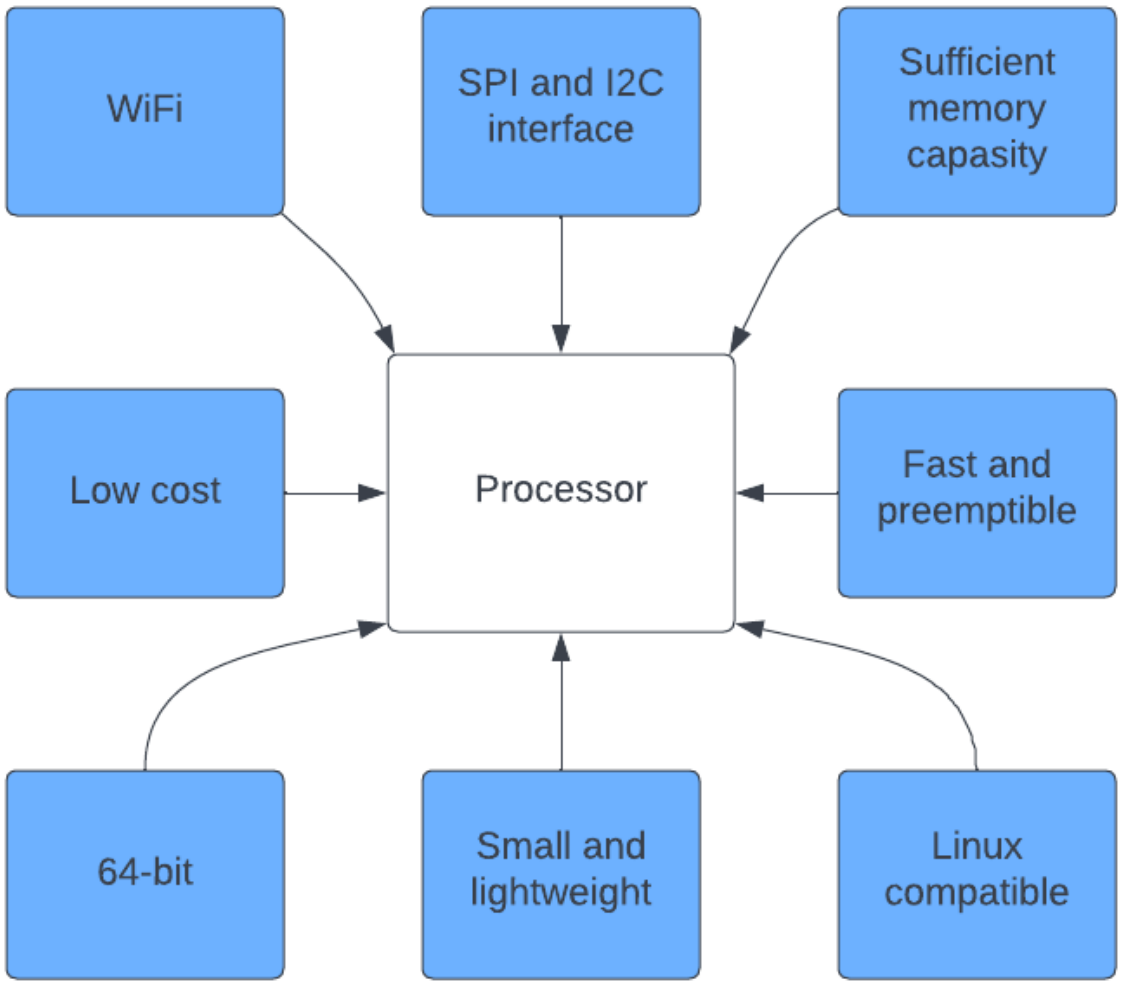


Figure 3.5: Microprocessor criteria

3.4 Modularity

The goal is to have a drone that is both easy to manufacture and has the ability to easily change parts when needed. This could be if there is a new technology or hardware equipment that would be better suited or if the drone needs repairs.

The end goal for this project is for UiA to have a self-developed operational drone swarm. Therefore, to continue a projection towards that goal it is critical that develop a solution that could support further development. There are also other aspects to consider when developing an operational drone swarm. Such as restricting the drone in size, mostly along the vertical axis. This is because the drone should have the ability to be stacked upon each other on a launcher.

The price point for each drone is not a high priority in this project, but is still taken into an account when choosing parts. The reason for this is that during the development of new solutions, it is critical with respect to time to not be restricted by the quality of the components. The best foundation would be laid by having parts that are more on-pair with the drone's needs, and after finding the criteria in the final drone iteration, the components can be down or upgraded depending on the research discoveries.

Chapter 4

Method

4.1 The planning and management process

The planning phase is an important step to master for a well-accomplished project. The process concerning this bachelor thesis started with all the group members brainstorming ideas and thoughts around the given assignment. The next step was to start filling in milestones and delivery dates in Jira software which allowed the group to systematically enter important dates and sub-tasks related to ordering parts and testing the drones. Jira is a tool for software projects which a road map, status on tasks, and a connection to GitHub for every group member to have access to both different codes and the current progress at all times.

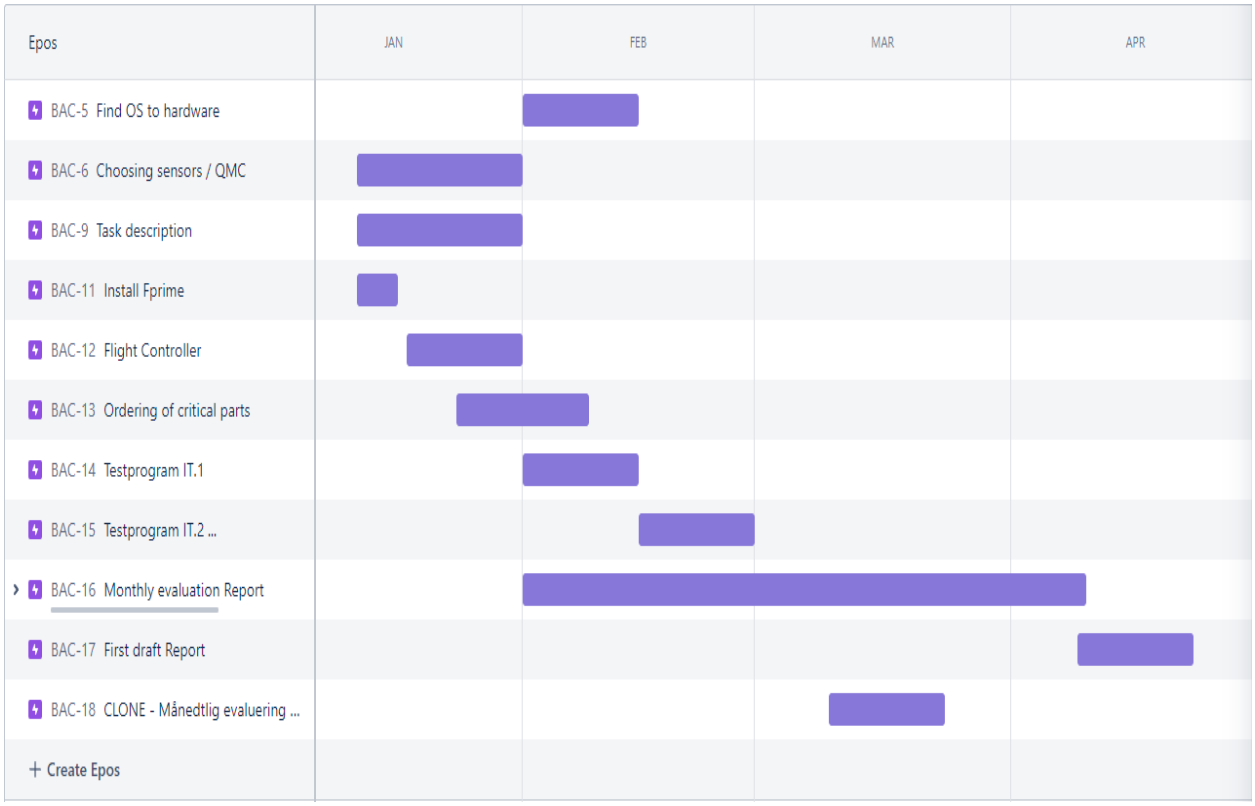


Figure 4.1: Example of how Jira software can be used, with how both the progress and different codes are available for everyone working on the project.

The management structure
There is no immediate group leader in this project, but it still exists a kind of hierarchy based on the different sub-projects. So the leadership position is a floating term based on what subject each person is working on.

4.2 Choosing microprocessor

The flight controller needs a microcomputer that will process all the data it receives and then act upon it. This microcomputer will work as the brain of the drone, and that is why it is important to choose wisely. By looking at the criteria in figure 3.5, a comparison between different microcomputers can be conducted.

Using these criteria, one can filter microcomputers and find the best candidates for the project. The best candidates are Raspberry Pi Compute Module 4 and Raspberry Pi Zero 2 W, Nvidia Jetson Nano, and ASUS Tinker Board S [69]. Then they are compared against one another based on the criteria given. This can be seen in Table 4.1 below. Data with no individual reference comes from the datasheets referenced at the name of the microcomputer.

Table 4.1: Table of criteria for microcomputer

Criteria	Onboard WiFi	64-bit Processor	SPI	I2C
RPi CM4 [33]	Yes	Yes	Yes	Yes
RPi Zero 2 W [48]	Yes	Yes	Yes	Yes
Nvidia Jetson Nano [93]	No	Yes	Yes	Yes
Asus Tinker Board S [13]	Yes	No[10]	Yes	Yes

Criteria	Size	Price	Weight
RPi CM4	55 x 40 x 4.5 mm	848,49 NOK [33]	12 g[58]
RPi Zero 2 W	65 x 30 x 5 mm	185,39 NOK [48]	11 g [30]
Nvidia Jetson Nano	69.6 mm x 45 mm	1449 NOK [70]	138 g [92]
Asus Tinker Board S	85.5 mm x 54 mm	1546 NOK [35]	55g [12]

Looking at the comparison above (Table 4.1) it can be seen that the ASUS tinker board does not have a 64bit processor, and as this is important to ensure enough processing power to complete the tasks fast enough, the ASUS tinker board will be ruled out. The NVIDIA Jetson also sticks out as it does not have onboard WiFi but after some quick research, it is found that a tiny USB-wifi adapter is the only thing needed to get WiFi[114]. Therefore the NVIDIA Jetson was still under consideration. Since weight and size are important due to the end goal, which is to reduce the weight of the drone, it quickly became clear that the weight of the NVIDIA Jetson is a problem. Two microprocessors are then left for consideration. The RPi CM4 and the RPi Zero 2 W. The two are very similar, from the same manufacturer and running the same processor. The differences are the size, the price, and the type of connection to the planned breakout board. The RPi Zero 2 W is much cheaper than the RPi CM4, at around a fifth of the cost.

An in-depth look into even more information given about the connectors on the boards reveals that the Zero 2 W has mounted connectors, and the GPIO signals are sent via holes for the mounting of pins. The CM4 is the opposite, with no pre-mounted connectors it is specialized for industrial use where a carrier board will be designed, and the designer can choose freely what connectors to mount and where. This would be better in the drone since it is possible to neglect the connectors that are not needed. That is also needed to design the carrier board so that the connectors will be in the right direction or position. Since the board is to be mounted inside the drone, the space is limited, and there might be positions that will fit a connector, and positions that will not. There will also be some positions that it will not be usable in.

The CM4 only has 2 HIROSE connectors(Section 4.7.3 for information on HIROSE) to connect it to the designed carrier board, whilst the Zero 2 W will need pins between the cards at the GPIOs that are in use. The form factors on the two boards are also slightly different as the Zero 2 W is longer and thinner, and the CM4 is shorter and wider. Since the drone has a somewhat square

room in the middle for components, it is better with the shorter, wider version. Due to the need for a specially designed form factor and connector placement on a carrier board the CM4 is therefore a better option, although it is more expensive. The previous project [79] also built a carrier board around the CM4, so CM4s are available at the university, which is helpful during the component shortage that makes delivery times longer (Section 6.1).

4.3 PCB V0 from the previous project

The PCB V0 is the version made by the previous bachelor project. This version is used with the RPi CM4 as a carrier board, but not as a flight controller. This version was used to communicate with a launcher and another drone. The previous drone used a Pixhawk 4 MINI as a flight controller. The Pixhawk is a fully completed flight controller which has preinstalled the required software and autopilot. PCB V0 has integrated HIROSE connectors to connect the RPi CM4, micro USB to connect to the RPi, UART connection, and a set of multi-purpose pins at J2. There were also integrated LEDs for power and activity on the RPi, and a voltage regulator to make sure the RPi always gets 5 volts. Further detail on the design and production of this is found in the previous bachelor thesis on the subject [79].

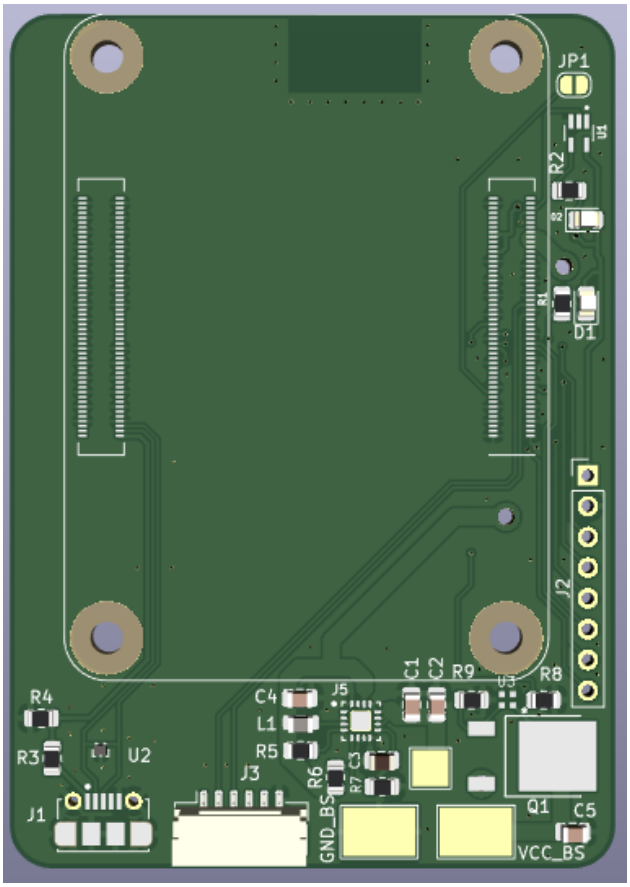


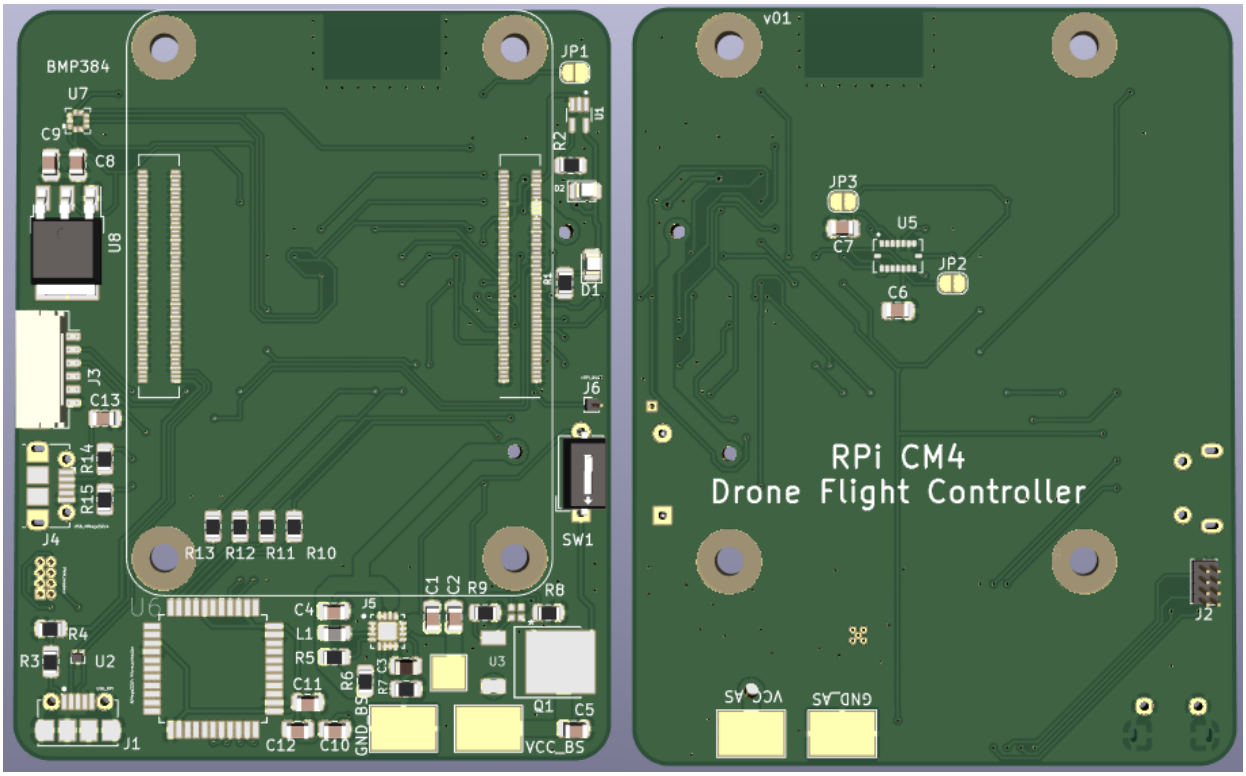
Figure 4.2: The previous version of the carrier board, referred to as V0

4.4 PCB V1

4.4.1 Design of new PCB

The design of a lighter integrated flight controller is critical for reaching the goal of a drone that is lighter than 250g. The drone from the previous project [79] was 320g, which was 70g too much for flying a drone outside without a license. Therefore it was important to make changes to cut weight. It was noticed that the previous version of the drone was using both an RPi with a carrier board to be able to connect to the launch pad used, and a Pixhawk 4 MINI as an autopilot [79]. The Pixhawk 4 mini is encapsulated in a metal box, which is then mounted inside the drone. This contributes to a lot of additional weight. With this setup, the drone also holds two onboard computers, which can be deemed unnecessary. It was therefore decided to put away the Pixhawk 4 MINI and focus on a carrier board for the RPi CM4 that would make the RPi able to act as a complete flight controller instead of just a support computer. Given the weight of the Pixhawk at 37g, the addition of components on the updated version of the carrier board would only be 2g, therefore it would save a lot of weight to do this.

It was also in the interest of the university to have a completely self-designed flight controller. By doing this it would not be necessary to spend large amounts on pre-made flight controllers, as a self-made one is much cheaper. This is especially important since the drones eventually are supposed to be used as swarm drones. When the Pixhawk is included in the drone it makes the stack of components inside the drone much taller. When the drones are to be put into a swarm, the plan is to use a launcher where the drones are stacked on top of each other. Therefore it is also in the project's interest to decrease the height of the drones so that more drones will fit on the same launcher. Based on these goals for the drone, it was decided to discard the Pixhawk and design a new flight controller. This will be done by further developing the previous carrier board for the RPi CM4 that is used in the drone [79], so that it includes the required sensors and PWM outputs to work as a standalone flight controller.



(a) Front (b) Back

Figure 4.3: Front and back of the first version of the further developed PCB carrier board

4.4.2 ATmega32U4 Microprocessor

Since the RPi CM4 only has 2 PWM channels [100] it is necessary for some sort of expansion that allows the system to control 4 separate PWM signals. The PWM also needs to always be prioritized as it is the signal that keeps the drone in the air, a separate microprocessor will be able to keep PWM at top priority. Due to this demand, it was decided to embed a microprocessor that could handle the PWM signal distribution. This microprocessor would receive the signal on how to distribute PWM via an SPI connection between the processor and the RPi.

There is a large variety of microprocessors available to use. There are too many to look at them all, so a handful of processors was picked to look further into and match with the criteria set by the project. Three processors were picked as they were listed as the best from multiple sources, such as an article by Adafruit on how to choose a microprocessor, [3]. The first processor mentioned is the AtTiny85, which is an 8bit processor. It is a very simple processor and with 8 bits it has its limitations with software. The chip features 6 I/O lines, which is not enough to connect with SPI to the RPi and distribute PWM signals at the same time, therefore it is ruled out [81]. The next chip is an ATmega328p, which is also 8bits, but is larger and features 23 GPIO connections [80], which is a lot more than the AtTiny85. The ATmega328p might therefore be a good choice, but as the last chip that was looked into is similar at much, the ATmega32U4 is slightly better, as it features even more GPIO connections, and has a great feature for built-in USB connectivity, making it accessible by USB. This feature is great for both programming and debugging the chip [14]. Due to this and the large supply issues with microprocessors at the time, the ATmega32U4 was chosen. It meets the requirements for the PWM distribution and is a much-used and well-documented microprocessor.

The ATmega would be embedded in the new carrier board for the RPi CM4 to ensure as few wires and movable parts as possible. The RPi has a large number of connectivity options, so the much-used I2C and SPI were evaluated before SPI was chosen [121]. This is due to the different transfer speeds the two different protocols offer, where SPI is twice as fast as I2C. Since the distribution of PWM signals has to be as fast as possible, SPI was the preferred connection protocol.

4.4.3 Embedding the ATmega32U4

There are several considerations to be taken when embedding the ATmega onto the carrier board. The decision to use SPI-connection has already been taken. The ATmega now needs to be connected to the right pins on the RPi CM4. The pinout schematic for the RPi CM4 can be found in the RPi datasheet, [100]. This will show where the pins from the ATmega should be connected. The pinout schematic that was used for the ATmega to ensure that the pins are connected in the right way can be found in the ATmega32U4 Datasheet[14]. First, a schematic is drawn to show connections, figure 4.4. After the schematic is drawn, it is inserted into the PCB design tool that is in use. The components are then placed appropriately, connectors on the edge of the board, decoupling capacitors as close to the microprocessor as possible, and then the tracks between them are routed. It is beneficial to avoid sharp corners on the tracks when routing. In figure 4.5 the components from the schematic in figure 4.4 are highlighted to show where they have been placed.

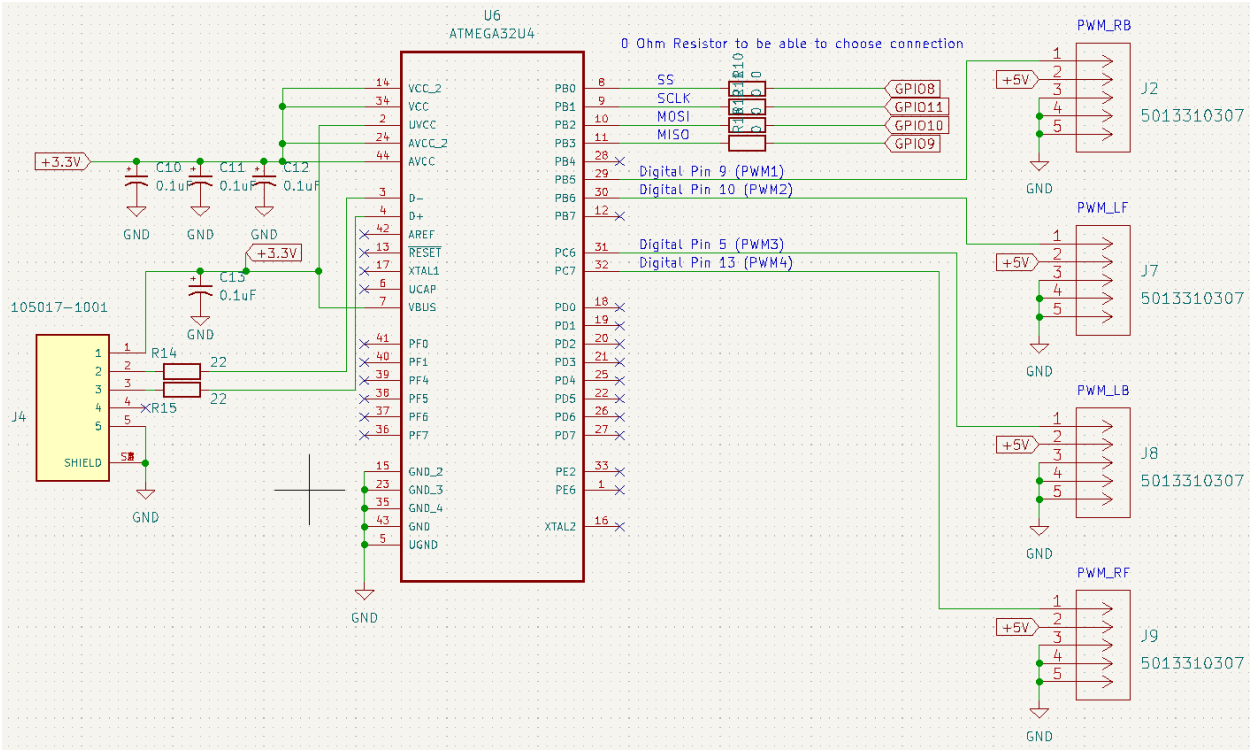


Figure 4.4: ATmega32U4 Schematic

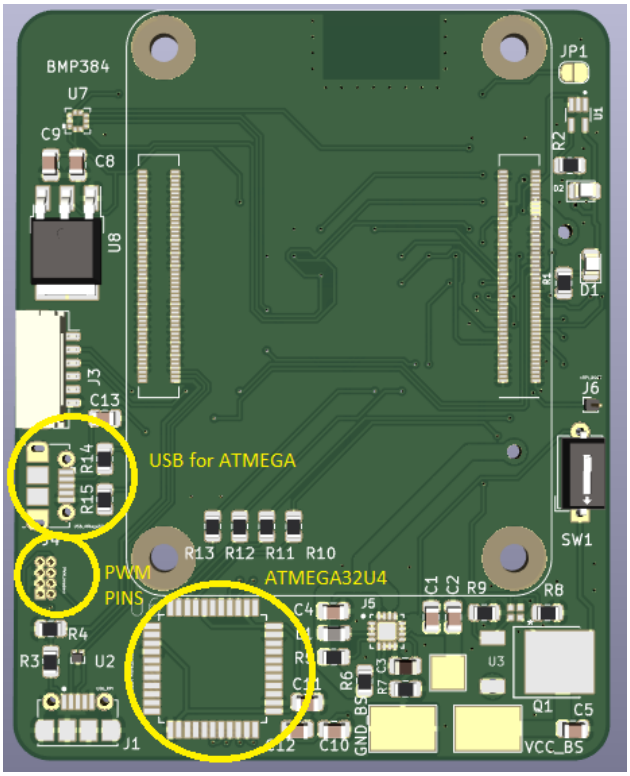


Figure 4.5: Front of carrier board highlighting ATmega and its connectors.

SPI connection to RPi

Pin 8,9,10 and 11 on the schematic above, figure 4.4, show the 4 connections needed to connect the ATmega to the RPi with SPI. They each have a resistor on 0 Ω so that the connection between RPi and ATmega can be chosen to not be active if these resistors are not in place.

USB connector

A USB connector is also hooked up to ensure connection directly to the microprocessor so that setup and debugging can be done more easily. According to the design requirements in section 21.5 in the ATmega Datasheet [14], the USB needs two 22 Ω resistors on D+ and D-, which are the data transferring wires of the USB. These resistors are fitted to save the chip in case of a shorting in the power supply circuit.

Decoupling capacitors

To ensure power to the chip, it is connected to the +3.3v line on the board. This line draws 3.3v from an output that the RPi supplies that are intended for components. The RPi regulates from 5v to 3.3v on the outputs intended for powering components. On the routing that supplies power to the chip, three decoupling capacitors have been fitted, each at 0.1μF. These are highly recommended in the design guidelines, section 21.5 in the ATmega datasheet [14]. Explanation of decoupling capacitors in section 2.1.4.

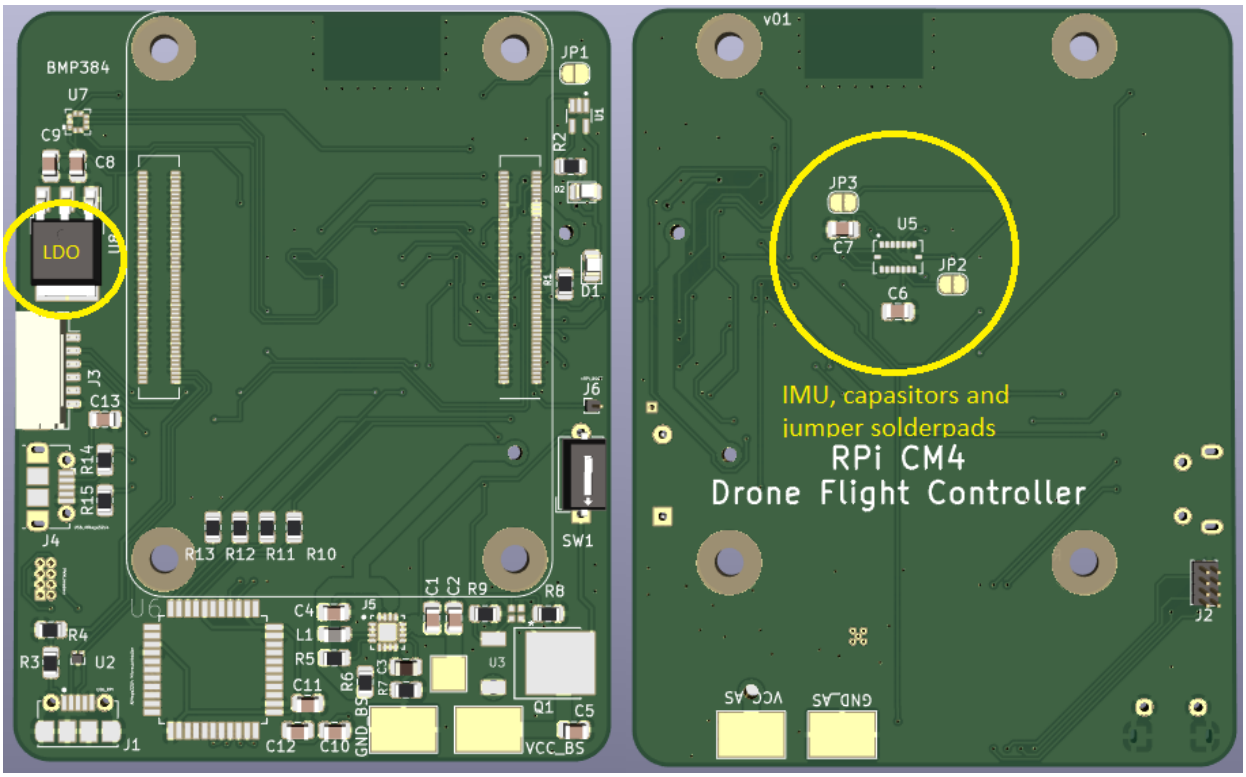
PWM Pins

Pin 29, 30, 31, and 32 are all separate PWM connections that are routed to an 8-Pin header (J2),where they make up 4 of the 8 pins, whereas the last 4 pins on this header are GND. These pins will emit the PWM signal that is used to control the 4 separate motor signals. This connector is highlighted as PWM Pins in figure 4.5.

4.4.4 Inertial Measurement Unit - IMU

An IMU is needed to control the drone while airborne, and there are several considerations to be taken into account when choosing the right IMU sensor. In this case, there are a lot of IMUs that would fit the drone and its criteria. Due to software encounters between the IMU and other components, a Bosch IMU is preferred. There is also a worldwide shortage of electrical components during the project, a large factor is therefore availability of the chosen sensor. After looking at the mentioned criteria, the choice landed on the BMI085 6-axis IMU from Bosch [23]. The choice would have landed on a BMI088 instead if it was available[25]. The differences between these two are small, but the wanted BMI088 is specially made for drones, so it has some features to increase robustness and vibration resistance [24], whereas the BMI085 is more specified for AR and VR implementations with features like platform stabilization for video and photo [22]. The BMI088 also has a higher range of measurements at 24G, whilst the BMI085 only has 16G. Although they are slightly different, they share the same footprint and pinout, meaning that it is possible to use the BMI088 at the BMI085 footprint on the PCB without changes, if one is acquired. Pinout diagram and footprint can be found in section 7.1 in both datasheets. [23] [25].

The IMU and its directly connected components, like the necessary decoupling capacitors and the solder jumpers, are located on the back of the PCB. As seen highlighted in figure 4.6. This has the advantage when it comes to the placement of the IMU. The sensor should preferably be as centered as possible. If located on the back it does not crash with the RPi CM4 that sits close to the PCB surface on the front side. It is also easier to move it slightly so that it is centered in a later version of the board. This will be addressed when the design for the drone body is ready.



(a) Front (b) Back

Figure 4.6: Front and back of PCB with highlighted IMU and LDO

4.4.5 Embedding the IMU

To embed the IMU it needs to be connected to a chip. In this case, it will be connected to the RPi using an SPI connection. A connection diagram for the SPI connection is provided in the datasheet from the manufacturer [23]. This is shown in the figure underneath (Figure 4.8). This connection diagram is then followed to ensure the right connection to the RPi. The figure underneath (Figure 4.7) shows the schematic for the BMI085 on the carrier board. The different GPIO pins are located on the RPi.

Low Dropout Voltage Regulator

Small sensors like the Bosch BMI085 are not particularly well equipped to deal with voltage drops on the power supply lines, because of this, the carrier board has been equipped with a Low Dropout Regulator 2.1.3 which ensures that the power that is supplied to the sensors has as little noise as possible. The power that has been through the LDO to ensure little noise is called 3.3V-LDO, and is the power supply to the sensor, as seen in figure 4.7 and highlighted in figure 4.6.

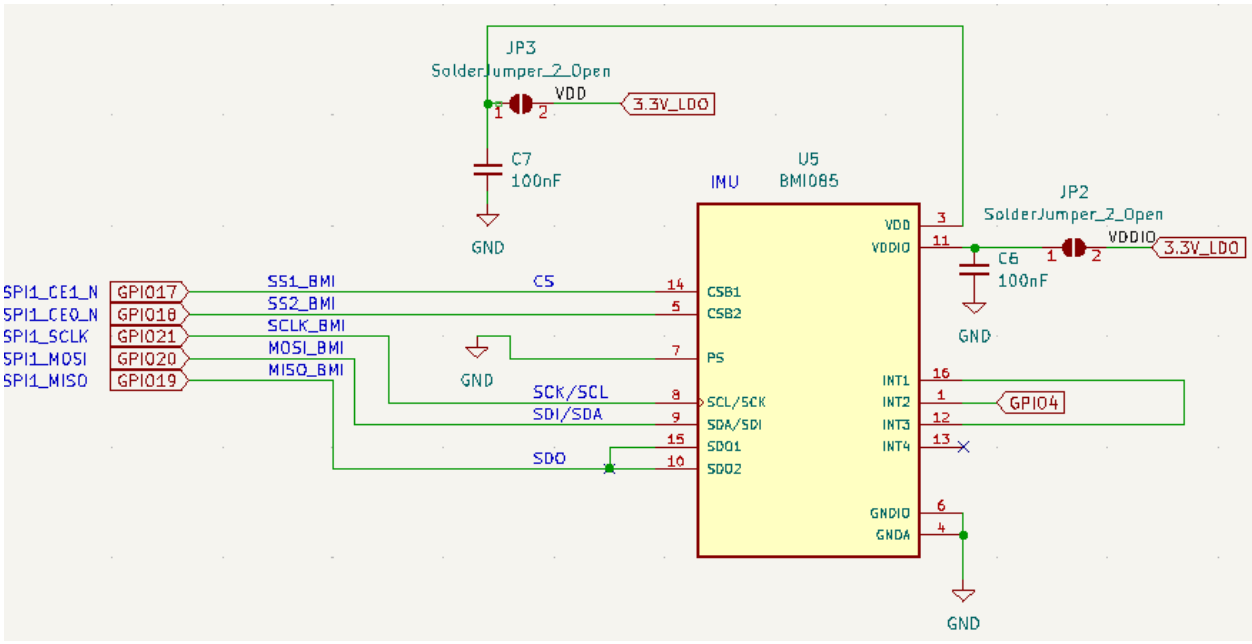


Figure 4.7: BMI085 Schematic

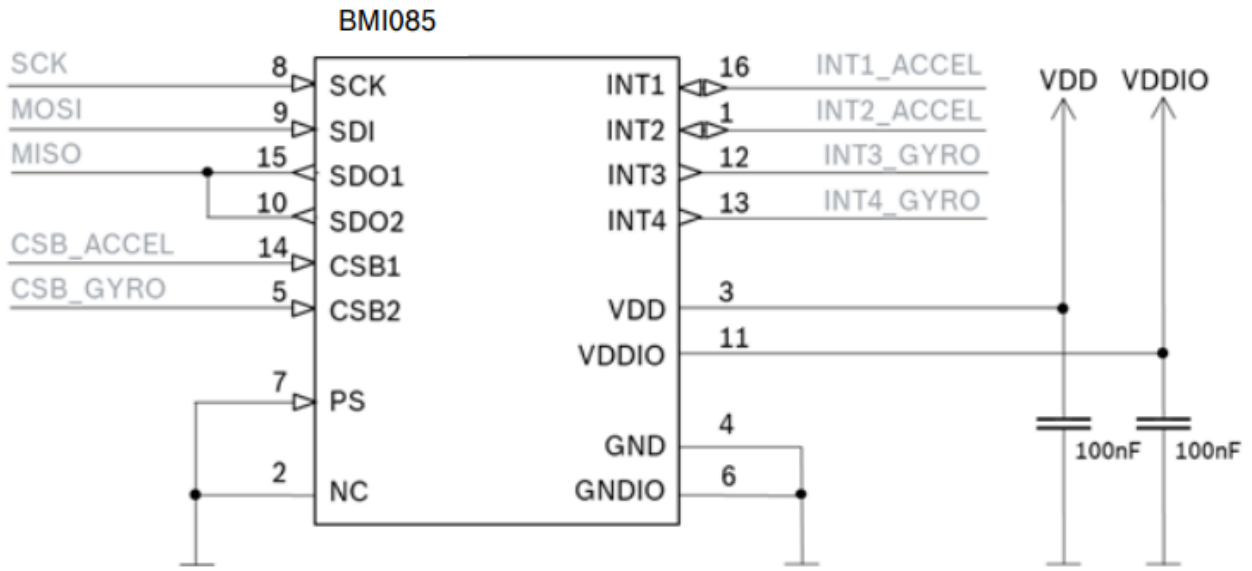


Figure 4.8: BMI085 SPI Connection Schematic, section 7.2 from BMI085 Datasheet [23]

4.4.6 Barometer

A barometer is also added so that the drone will be able to get sensor data that measures the atmospheric pressure, thus letting it be able to know how high above the ground it is. A barometer might need re-calibration from time to time, as the environment around the drone changes, so software utilities are needed to be able to re-calibrate it. This will be added to the software that the drone is running. The barometric pressure sensor that was chosen was the BMP384 from Bosch, it works well with other Bosch sensors such as the chosen IMU. Originally the BMP388 is better suited as it is specially made for drone applications, and it is a good fit with the IMU, but the BMP388 was impossible to get a hold of due to the previously mentioned shortage. Therefore the BMP384 was chosen, as it shares a lot of the specifications with the 388 and is also available. It is a small, robust sensor that will fit well into drone applications.

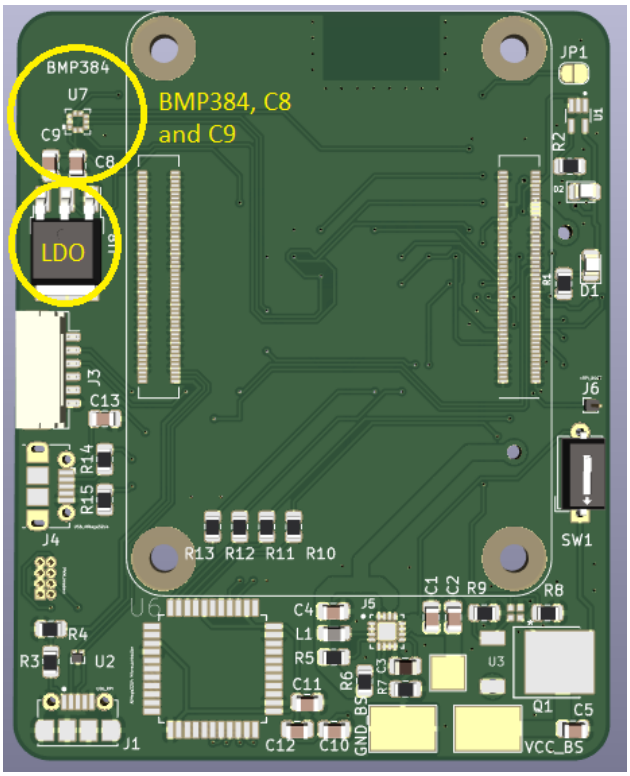


Figure 4.9: BMP384, LDO, decoupling capacitors C_8 and C_9

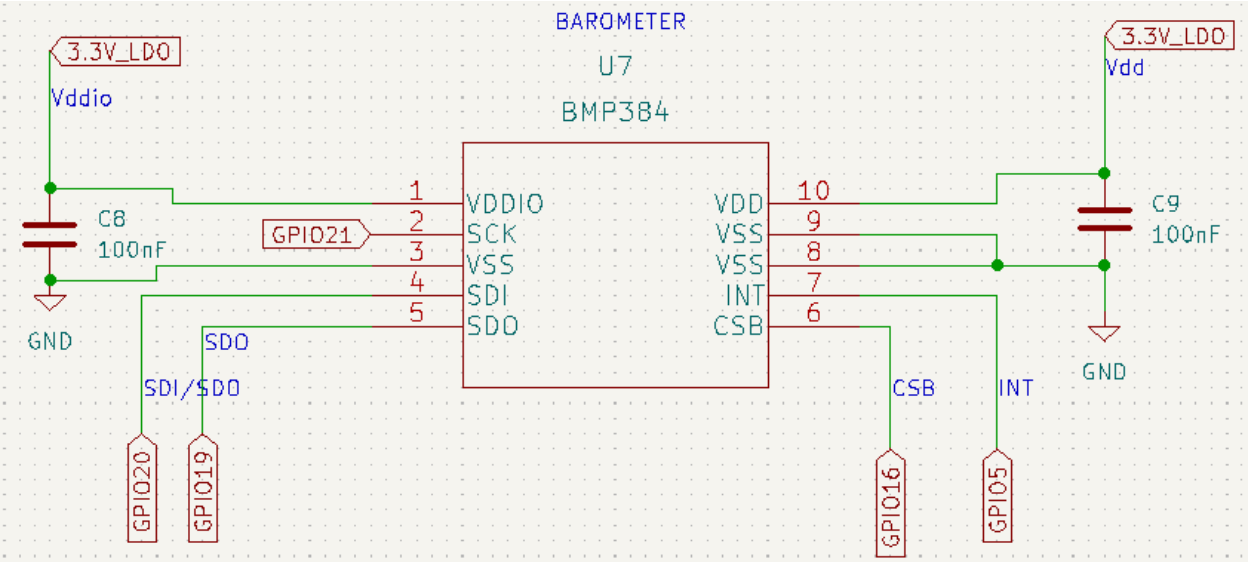


Figure 4.10: BMP384 Schematic

4.4.7 Embedding the barometer

Firstly the BMP384 is included in the schematics as seen in figure 4.10 above. Here it is connected to the rest of the circuit. Two decoupling capacitors are also added, both at 100nF, according to the SPI schematic (Figure 4.11) in section 6.2 in the BMI384 Datasheet. A note is added to the SPI schematic in the datasheet that specifies the value of C_1 and C_2 (C_8 and C_9 in figure 4.9)[26]. The BMP384 is connected to the 3.3v-LDO power line to ensure as little noise as possible. It is also connected to the RPi using the SPI connections. This can be seen in the schematic (Figure 4.10) and is done following the SPI schematic in figure 4.11 from the datasheet.[26]

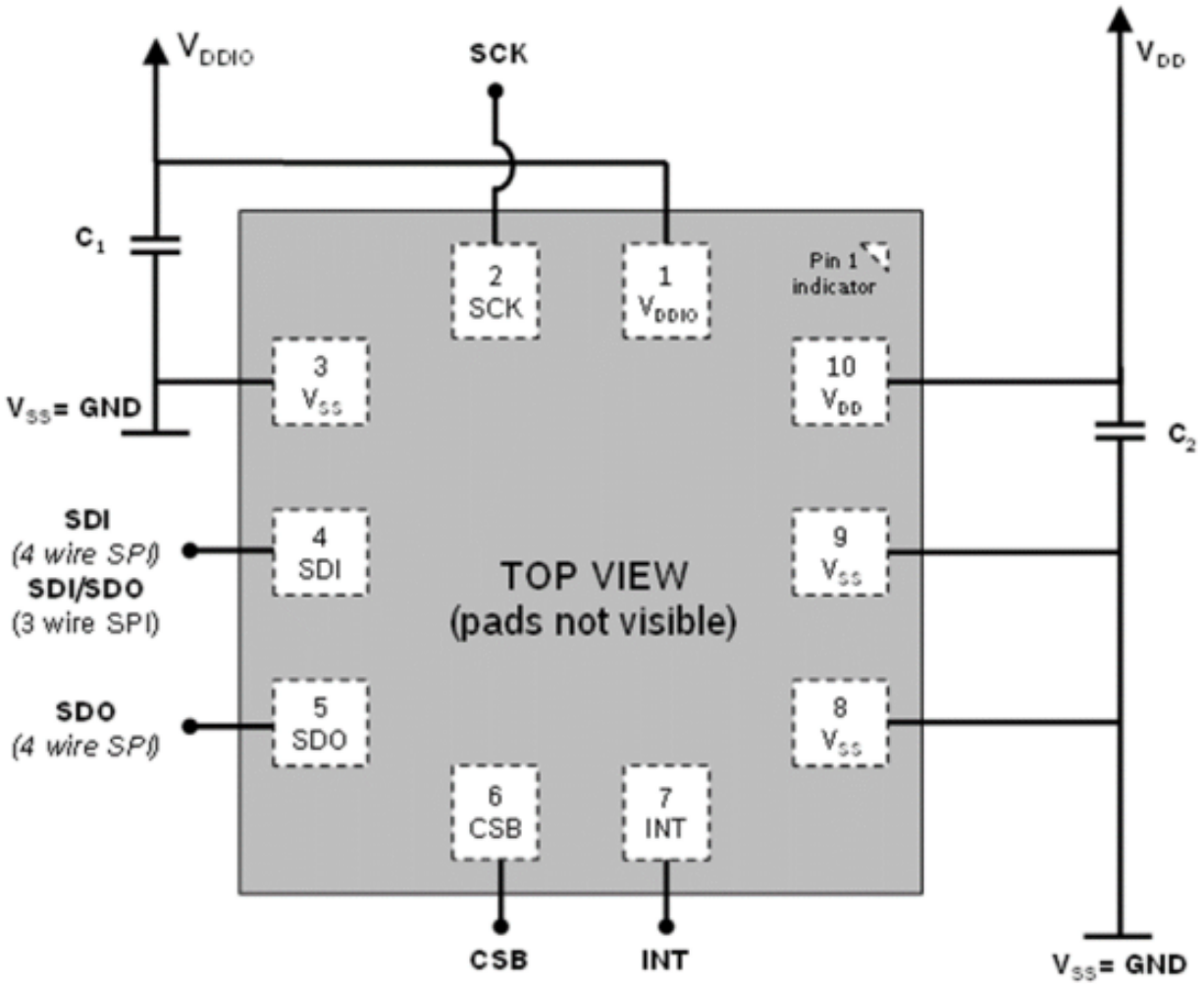


Figure 4.11: BMP384 SPI Schematic from The BMP384 Datasheet [26].

4.4.8 Power switch possibilities

Although the schematic (Figure 4.13) for a switch and its pads was already in place at v0 from the previous project [79], the switch and its tracks were not placed on the PCB. Therefore a switch was added to the PCB and it was routed from the appropriate pad to the step-down converter.

The possibility to turn on and off power to the whole board with a switch is a handy function to have, this can be used to ensure that the board stays off and does not draw power from the battery when it is supposed to be off. It can also be used to power off the board in case of a system freeze, although this is never recommended as it may cause permanent damage to memory if the board is in the middle of a crucial task.

The switch is added as an alternative so that if a switch is deemed necessary. The power is connected to the VCC_{BS} pad which is highlighted in yellow in figure 4.12. The power will then have to travel through the switch before it goes to the step-down converter (Section 4.4.10).

The switch in question is in a slightly large packet, therefore if the switch is deemed unnecessary, the power can simply be connected to the smaller pad, highlighted in black in figure 4.12, and it will bypass the switch and go directly into the step-down converter, so the mounting of the switch can be left out.

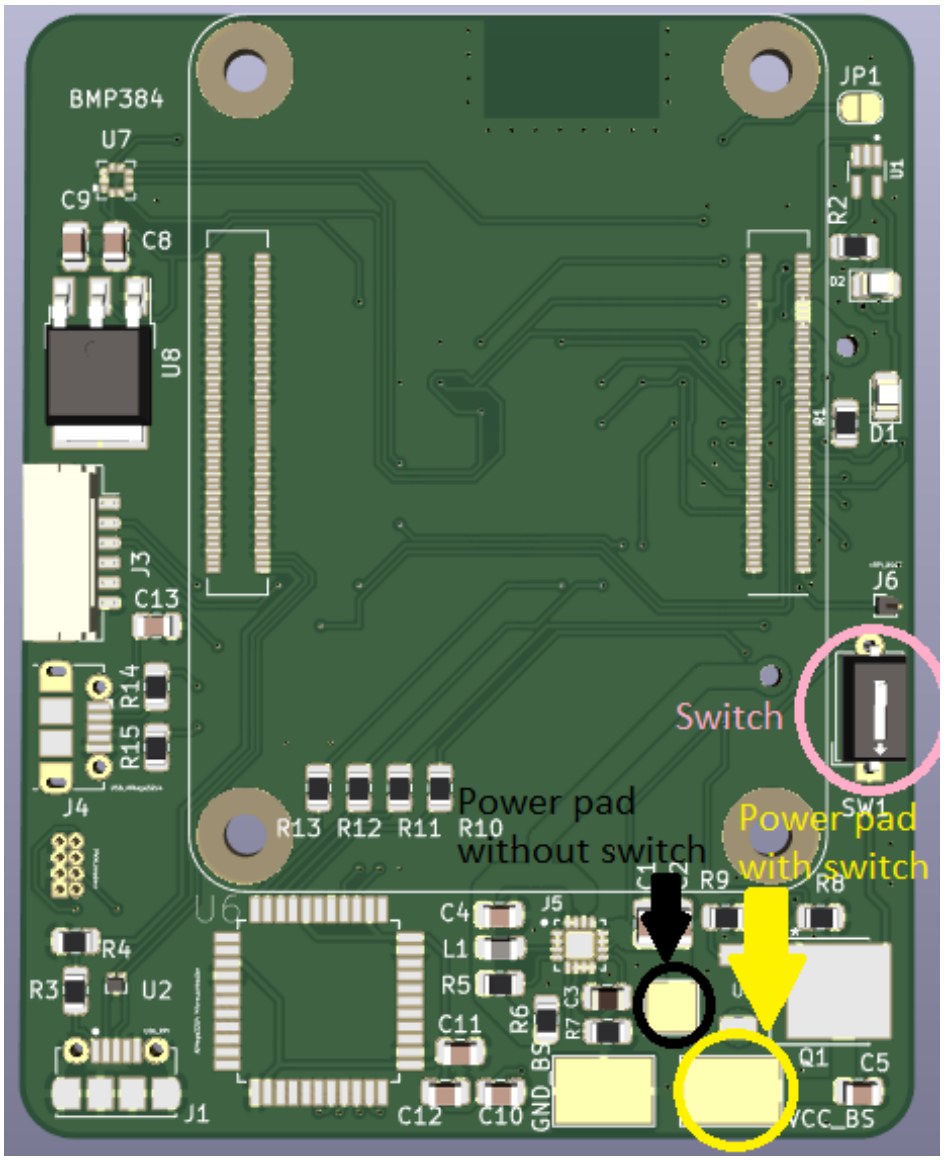


Figure 4.12: PCB with highlighted switch possibilities

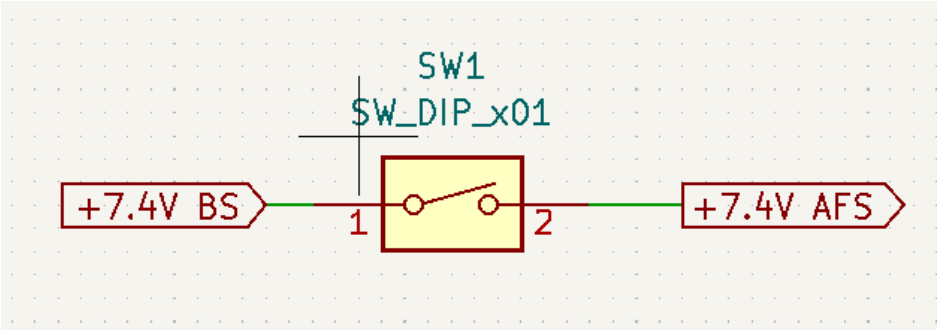


Figure 4.13: Schematic for the onboard switch, BS is before the switch, and AFS is after the switch

4.4.9 Outer measurements and increase in size and weight

While fitting the footprint for the ATmega (Section 4.4.2) and moving the JST GH connector, it was decided to expand the left side of the board slightly. The area to the left of the left side mounting holes was expanded from 4.5mm in iteration v0, to 10.53mm in V1, increasing the total width of the board by 6.03mm. As this was done early in the process, it did not make a large difference to the mechanical design of mounting brackets, as these were not made yet. Important criteria for the PCB is to keep a small size, therefore it was not increased by much, so as seen in V2 and V3 the left side still gets a tight fit. The measurements of the board and key features like center and mounting holes can be seen in figure 4.14 underneath. The weight of the V0 carrier board is 13.8 grams, and the updated version, V3 weighs 15.8 grams. Giving an increase at only 2 additional grams to fit the components needed to get rid of the 37 grams heavy Pixhawk.

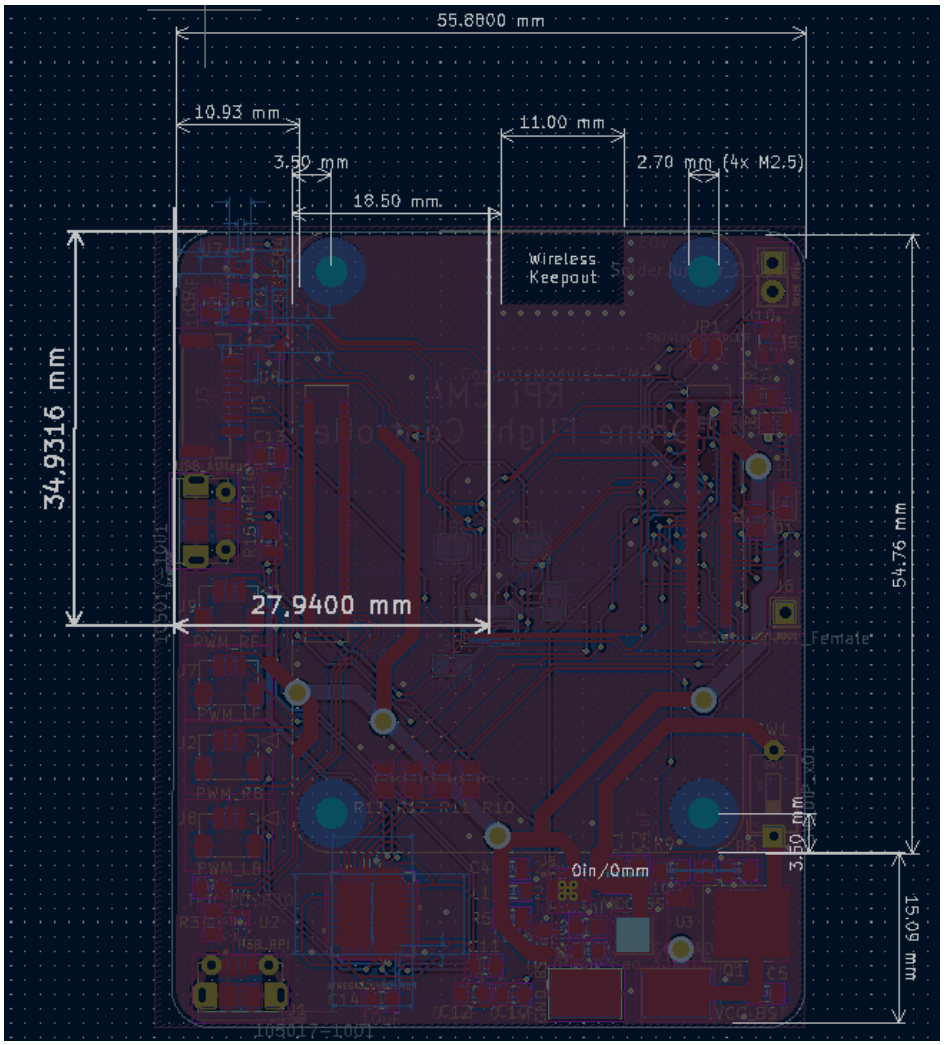


Figure 4.14: PCB measurements for PCB V1 to V3

4.4.10 Total power consumption of the system

In addition to the RPi, the ATmega, the barometer, the IMU and the LEDs are drawing power on the carrier board. Using the datasheets for the selected components it can be found how much current the components draw. This is partly done for the IMU and barometer in section 4.5.4, where it is needed to calculate if the LDO can supply enough current. Therefore to check power consumption for the whole system, the results from IMU and barometer are added to the rest of the power-consuming components.

Each LED consumes $0,2269mA$ [68] and the ATmega32U4 can at maximum speed and voltage consume $27mA$ [14]. The IMU and barometer consume a maximum of $5.1534mA$ combined. See section 4.5.4 for more on the calculation of the IMU and barometer.

Since

$$Supply_{Total} = I_{IMU+Barometer} + I_{LED} * 2 + I_{ATmega}$$

then

$$Supply_{Total} = 5.1534mA + 0.4538mA + 27mA = 32.6072mA$$

The voltage regulator from Texas Instruments that is used in V0 [79] is able to supply $3A$ [55], therefore it is still more than sufficient to supply the updated versions up to V3, since it still only requires $32.6072mA$. The extra power is not needed at the moment but as ESCs might draw power directly from the carrier board in the future, $3A$ might be necessary to power four ESCs.

4.4.11 Raspberry Pi GPIO Pin Configuration

The Raspberry Pi Compute Module 4 has a set of multi-purpose GPIO pins available through the connection to the carrier board. These pins can individually be set to different alternatives. In figure 4.15 below the different alternatives are listed in a table. Each pin has six different alternatives from ALT0 to ALT5. These alternatives have to be set in the software that the RPi is running. To make the GPIO pins receive and understand the right signal from the IMU and barometer, the pins GPIO14-GPIO21 have to be set to ALT4. This will set these GPIO pins to SPI Protocol, making the signal output from the sensors readable for the RPi.

2.5.1. Alternative Function Assignments

Up to 6 alternate functions are available. The BCM2711 ARM Peripherals book describes these features in detail. The table below gives a quick overview.

Table 1. GPIO Pins Alternative Function Assignment

GPIO	Pull	ALT0	ALT1	ALT2	ALT3	ALT4	ALT5
GPIO0	High	SDA0	SA5	PCLK	SPI3_CE0_N	TXD2	SDA6
GPIO1	High	SCL0	SA4	DE	SPI3_MISO	RXD2	SCL6
GPIO2	High	SDA1	SA3	LCD_VSYNC	SPI3_MOSI	CTS2	SDA3
GPIO3	High	SCL1	SA2	LCD_HSYNC	SPI3_SCLK	RTS2	SCL3
GPIO4	High	GPCLK0	SA1	DPI_D0	SPI4_CE0_N	TXD3	SDA3
GPIO5	High	GPCLK1	SA0	DPI_D1	SPI4_MISO	RXD3	SCL3
GPIO6	High	GPCLK2	SOE_N / SE	DPI_D2	SPI4_MOSI	CTS3	SDA4
GPIO7	High	SPI0_CE1_N	SWE_N / SRW_N	DPI_D3	SPI4_SCLK	RTS3	SCL4
GPIO8	High	SPI0_CE0_N	SD0	DPI_D4	BSCSL / CE_N	TXD4	SDA4
GPIO9	Low	SPI0_MISO	SD1	DPI_D5	BSCSL / MISO	RXD4	SCL4
GPIO10	Low	SPI0_MOSI	SD2	DPI_D6	BSCSL SDA / MOSI	CTS4	SDA5
GPIO11	Low	SPI0_SCLK	SD3	DPI_D7	BSCSL SCL / SCLK	RTS4	SCL5
GPIO12	Low	PWM0_0	SD4	DPI_D8	SPI5_CE0_N	TXD5	SDA5
GPIO13	Low	PWM0_1	SD5	DPI_D9	SPI5_MISO	RXD5	SCL5
GPIO14	Low	TXD0	SD6	DPI_D10	SPI5_MOSI	CTS5	TXD1
GPIO15	Low	RXD0	SD7	DPI_D11	SPI5_SCLK	RTS5	RXD1
GPIO16	Low	<reserved>	SD8	DPI_D12	CTS0	SPI1_CE2_N	CTS1
GPIO17	Low	<reserved>	SD9	DPI_D13	RTS0	SPI1_CE1_N	RTS1
GPIO18	Low	PCM_CLK	SD10	DPI_D14	SPI6_CE0_N	SPI1_CE0_N	PWM0_0
GPIO19	Low	PCM_FS	SD11	DPI_D15	SPI6_MISO	SPI1_MISO	PWM0_1
GPIO20	Low	PCM_DIN	SD12	DPI_D16	SPI6_MOSI	SPI1_MOSI	GPCLK0
GPIO21	Low	PCM_DOUT	SD13	DPI_D17	SPI6_SCLK	SPI1_SCLK	GPCLK1
GPIO22	Low	SD0_CLK	SD14	DPI_D18	SD1_CLK	ARM_TRST	SDA6
GPIO23	Low	SD0_CMD	SD15	DPI_D19	SD1_CMD	ARM_RTCK	SCL6
GPIO24	Low	SD0_DAT0	SD16	DPI_D20	SD1_DAT0	ARM_TDO	SPI3_CE1_N
GPIO25	Low	SD0_DAT1	SD17	DPI_D21	SD1_DAT1	ARM_TCK	SPI4_CE1_N
GPIO26	Low	SD0_DAT2	<reserved>	DPI_D22	SD1_DAT2	ARM_TDI	SPI5_CE1_N
GPIO27	Low	SD0_DAT3	<reserved>	DPI_D23	SD1_DAT3	ARM_TMS	SPI6_CE1_N

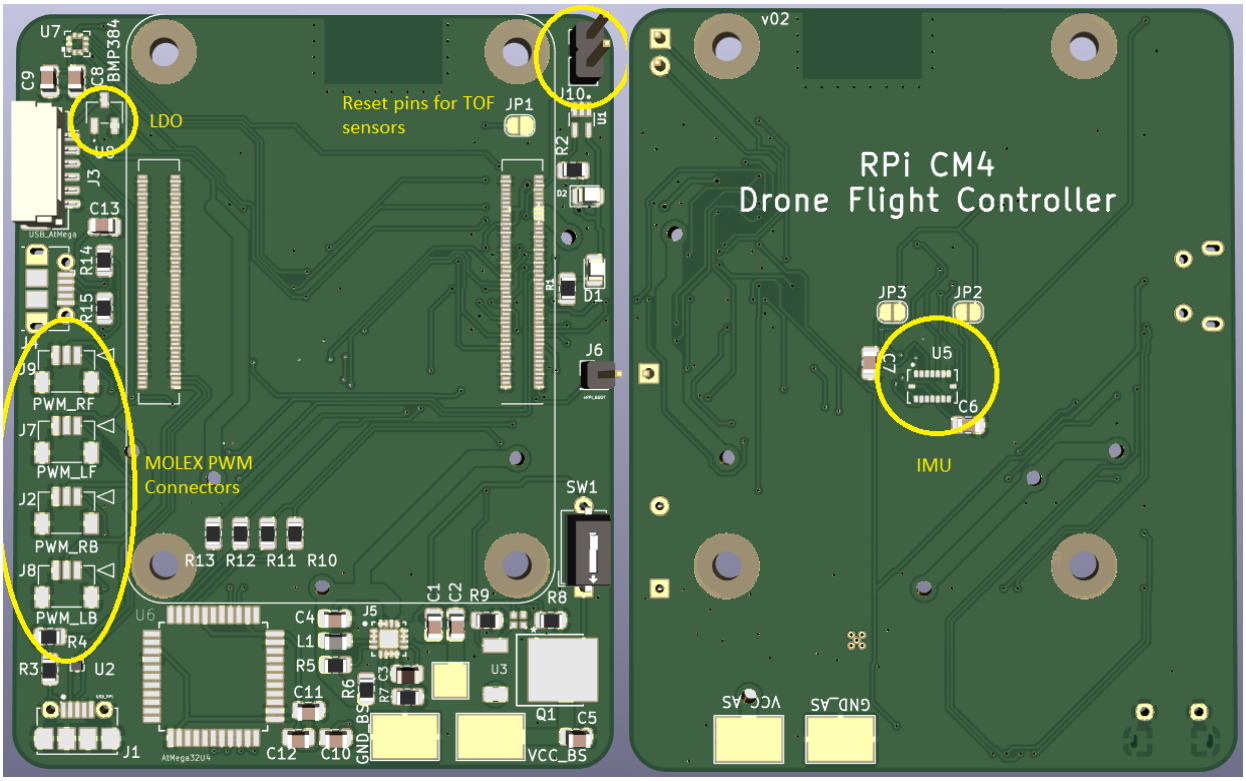
Figure 4.15: Alternative configurations for GPIO pins on RPI from the datasheet [100]

4.4.12 Calculation of track width

The track width on the carrier board varies from what signal the track is supplying. There are no changes in the highest voltage that has been running on the board since the last bachelor thesis on the project, therefore the same rules to the track width that was introduced there were followed. It concludes with smaller tracks for signals, and wider tracks for 5v, as higher voltage requires wider tracks. See section 5.3.3 in the previous bachelor thesis [79].

4.5 PCB V2

Upon ordering and receiving PCB V1, it was quickly uncovered that there would be necessary with an updated version to fit newly surfaced criteria in form of needed connections and faults with V1. Underneath is a further description of updates from V1 to V2. The updated version can be seen in figure 4.16 below.



(a) Front (b) Back

Figure 4.16: Front and back of PCB with highlighted changes from V1 to V2

4.5.1 Molex PWM connectors

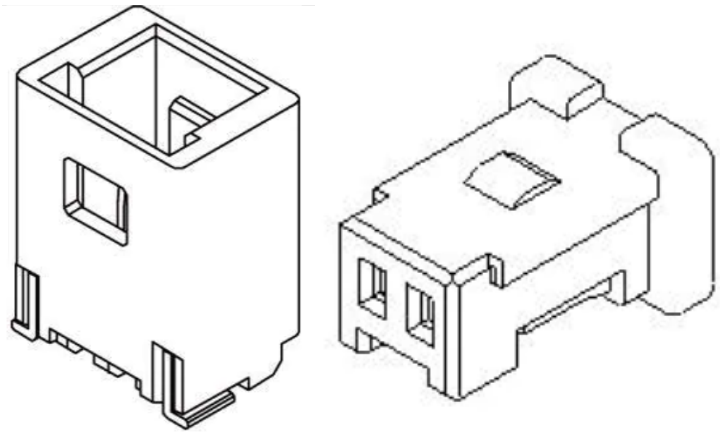
Choosing and fitting the connector

Upon receiving the PCB V1 it was quickly noticed that the PWM header that was supposed to distribute the PWM signals from the ATmega was too small. It was made with a 1mm pin size and should have been made with a 2.54mm pin size to fit regular wires and equipment. Although it would have been sufficient enough to change to a bigger 2.54 mm header, it was rather decided that to minimize the risk of wrong connection between motor and PWM signal header, should rather be implemented 4 separate contacts that would be impossible to put the wrong way, and they should be marked as Left Front(LF), Right Front(RF), Left Back(LB) and Right Back(RB). This will make the installation of motors to the carrier board easier and less likely to connect the wrong motor to a specified PWM signal.

Since the left side of the carrier board is closer to the ATmega it was decided to put the connectors there. This would mean easier routing from the microprocessor to PWM headers, but it introduced a challenge fitting 4 separate connectors given its limited space. Although there are a massive variety of different connectors, the shortage of components in the market made it difficult to find small enough connectors that were available. The criteria for the chosen connector were small size, with a preferred pin size of 1mm or 1.25mm, and the availability of such a connector.

The criteria led to the decision to use a 3 pin Molex Pico-Clasp connector, where both male [85], female [84], and lightning connector [83] for the cables were available. The size of the pins on the

connector is 1mm. Upon moving the barometer, the micro-USB for the ATmega, and the I2C connector higher uponthe left side of the board, there was enough space to fit the 4 separate connectors in a row as shown in figure 4.16, highlighted as Molex PWM connectors.



(a) Male (b) Female

Figure 4.17: Male, [85], and female, [84], Molex connectors

Molex Lightning Connector

The connectors additionally use a Molex Lightning Connector (Figure 4.18) on the end of each cable that is mounted to the connector. The cable is fitted in the small metal connector, and a pair of crimping pliers is used to press the connector on the wire. The wire is then inserted into the connector and clipped in place.

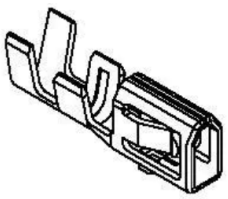


Figure 4.18: Molex Lightning Connector to fit at the end of connected cables

Connection schematic

In figure 4.19 below, the connections between the four Molex connectors are shown. The connector is a 3-pin connector, so pin 1 is connected to the PWM signal output from the ATmega microprocessor, pin 2 is connected to the 5v line, and pin 3 is grounded. Pin 4 and 5 are the shield on the connector and are therefore also connected to ground. On the carrier board pin 1 is located to the right, pin 2 in the middle and pin 3 on the left. The motors in use are currently receiving their power from the Holybro PM06 PDU (Power Distribution Unit) and do therefore not require 5v power through the Molex connector, so this will not be connected. The 5v line connection is put in place in case of a future motor swap, where a motor might need to receive its power from the carrier board. Since the motor and PDU setup is from a previous iteration of this project further elaboration on motor and PDU selection can be seen here [79].

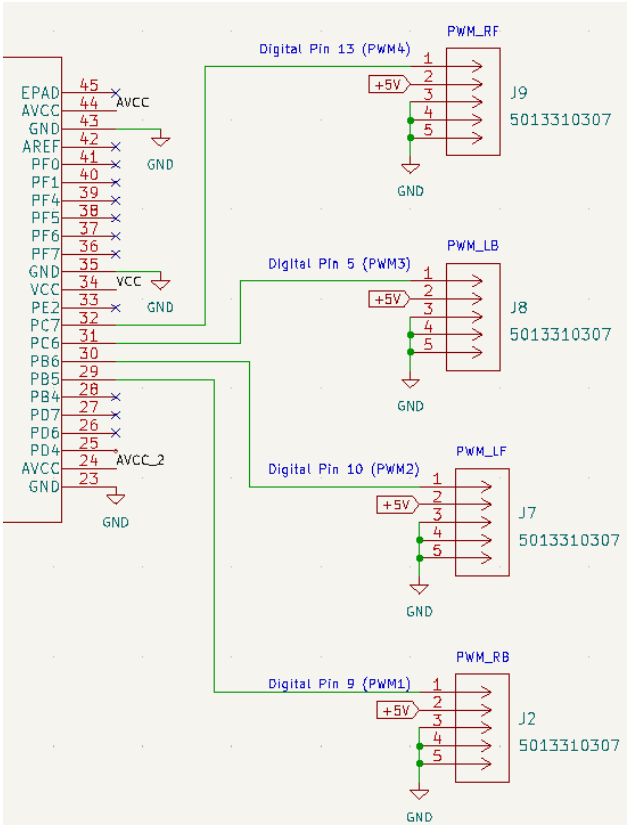


Figure 4.19: Four Molex connectors connected to ATmega microprocessor

4.5.2 Reset pins for ToF Sensors

Time of Flight (ToF) sensors are also to be added to the drone. These sensors are using the I2C communication protocol. The sensors are delivered with the same I2C address, and they need individual addresses to be able to register which sensor the signal is coming from. Therefore a different address has to be assigned when they start, and when turned off this change of address is reset. It is therefore necessary with one separate reset pin for each ToF sensor, in the first iteration of the ToF sensor addition is going to be added two sensors, therefore there are also added two reset pins. The two reset pins are connected to GPIO2 and GPIO3 on the Raspberry Pi. (Figure 4.20)

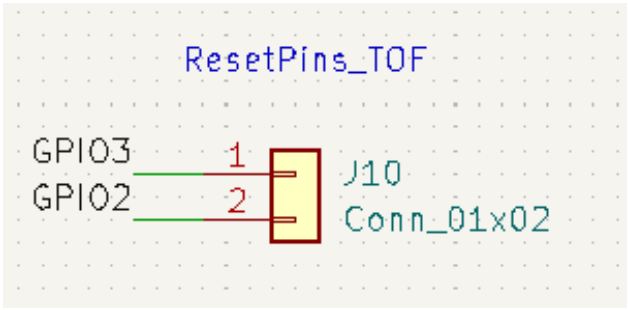


Figure 4.20: Reset pins for ToF sensor, GPIO pins on RPi.

4.5.3 Centered IMU

The IMU that is mounted on the backside of the carrier board needs to be either in the middle of the drone or otherwise compensated for its offset from the middle through software. If the possibility is there, the best and easiest way is to center the IMU on the carrier board. Therefore the center of the board was measured to move the IMU and its components in V2 of the board. There were some software limitations to find the center. The software uses a grid-based component movement system, so it was not possible to get the IMU dead center, but since the grid was adjustable, the offset from the center is only 0.0066mm in the y-direction and 0 in the x-direction. (Center in y at 34,925 and x at 27.94)

4.5.4 Change of LDO

On the first iteration of the carrier board an LDO was used to lower the noise on the power supply for the sensors. When V1 arrived it was noticed that the LDO was hooked up to 3.3v in and 3.3v out. This is not possible as an LDO needs to have additional voltage to regulate downwards. Additionally, it was noticed that the specific LDO used was much larger than necessary, therefore the LDO was changed. The LDO chosen is the AP2138n-3.3TRG1 and has the ability to supply $250mA$ at the fixated level of 3.3v, [54]. The barometer only needs 3.4 μA at 1 Hz [26] and the IMU needs 150 μA for the accelerometer in normal mode and $5mA$ for the gyroscope in normal mode [23].

Since

$$3.4\mu A = 0.0034mA$$

and

$$150\mu A = 0.15mA$$

then

$$Supply_{IMU+Barometer} = 0.0034mA + 0.15mA + 5mA = 5.1534mA$$

This means that the available $250mA$ is more than enough to supply the sensors with the power needed. The updated LDO can be seen highlighted in figure 4.16. Since the left side of the board was starting to fill up, it would be beneficial to choose an LDO with a small packet as well. The LDO chosen is a SOT 23-3, where SOT stands for Small Outline Transistor. It is a 3-pin LDO with 5v in, 3.3v out, and ground. The schematic for the connection of the LDO can be seen underneath in figure 4.21.

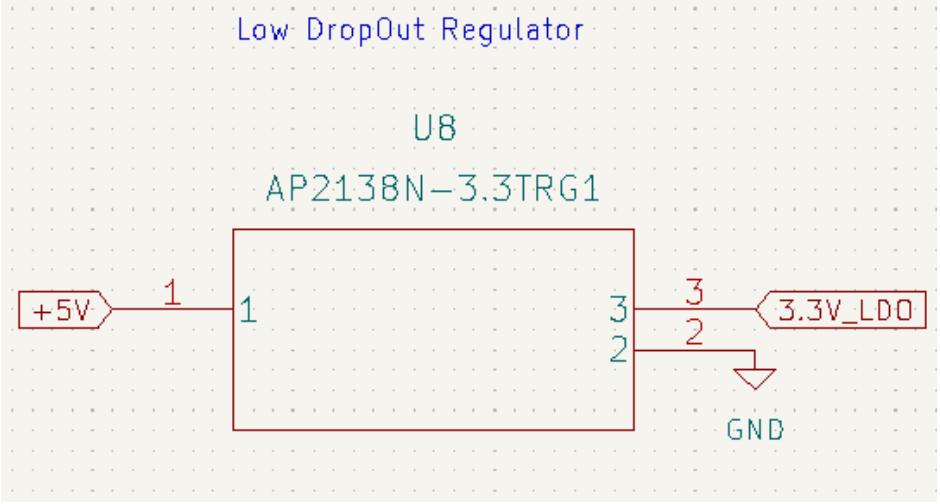
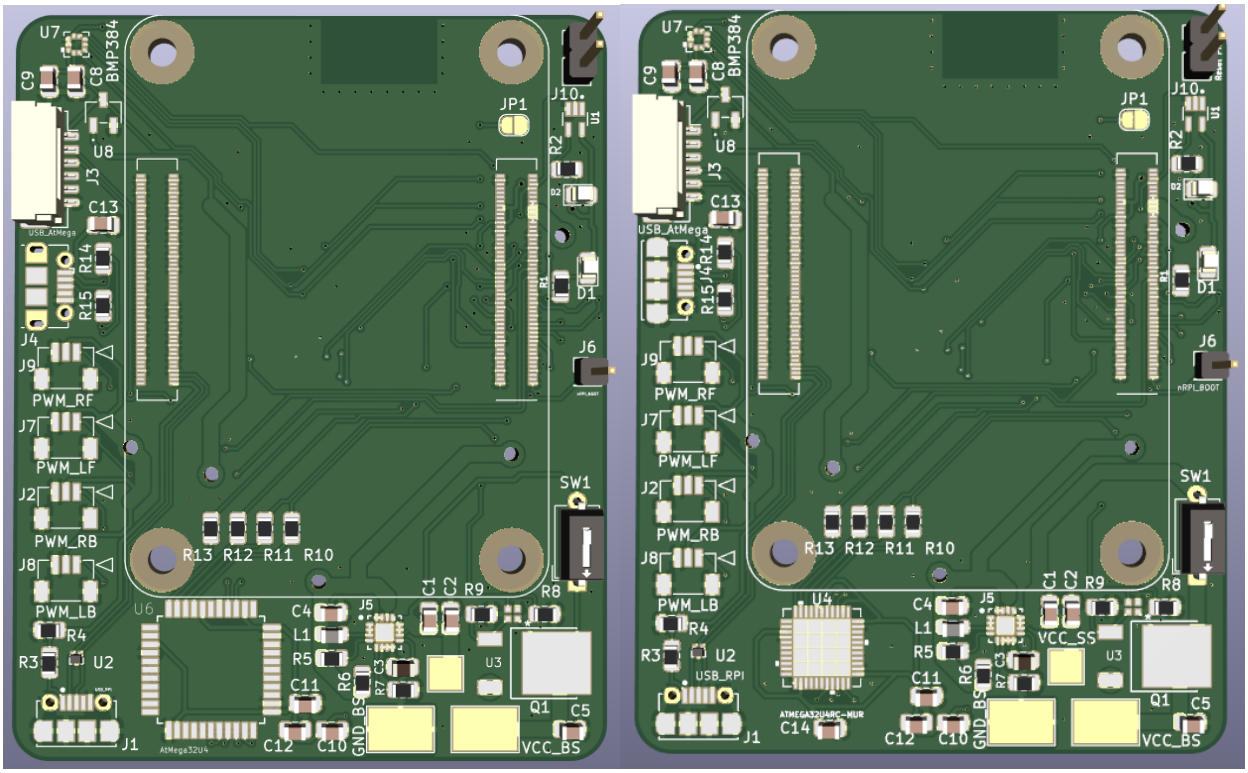


Figure 4.21: Schematic for connection of LDO AP2138N-3.3TRG1

4.6 PCB V3

4.6.1 Changing ATmega32U4 footprint

Upon the arrival of V2 and other components that were ordered there was a problem when mounting the ATmega32U4. The ATmega32U4 comes in different packets referred to as AU and MUR. The only difference between the two types is the package, where the size and therefore the placement of the pads in the footprint is different. Figure 4.22 underneath shows the difference in footprints. While using the footprint of the ATmega32U4-AU, which has pins sticking out from the packet, the ATmega32u4-MUR was ordered, this one does not have the pins, and is therefore smaller, and does not fit the footprint, [14]. That is why the footprints were swapped from AU to a MUR variant.



(a) V2 (b) V3

Figure 4.22: The difference in footprint for ATmega32U4 on the bottom of the board.

4.6.2 Changing JST GH connector from UART to I2C for ToF sensor

On the V2 of the carrier board the white JST GH 6 pin connector was connected to the RPi CM4 using UART. As this connector was intended to be able to be used for different sensors that might be needed later, it has now been dedicated to be connected to the ToF sensors that will be implemented. These sensors use I2C to communicate, and since I2C also is used on the majority of other sensors that might replace the ToF in the future it was decided to swap the connector from UART to I2C. Datasheet for the JST GH connector can be seen here [63]. To change from UART to I2C it is only necessary to disconnect the Rx and Tx wires, and then connect SDA and SCL to the correct GPIO pins from the RPi. This can be seen underneath in figure 4.23.

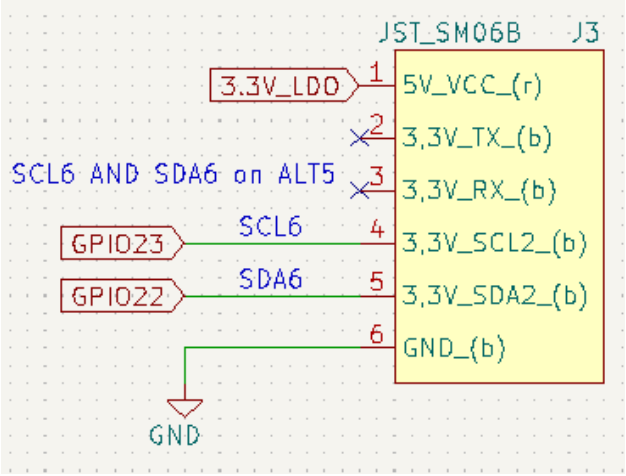


Figure 4.23: 6-pin JST GH connector [63]

4.6.3 Track width changes

Since the switch detailed in section 4.4.8 was not in use during testing in the first iteration, V1 of the carrier board it was not noticed immediately that the track width to and from the switch was too thin. It is supposed to lead 7.4v and it is therefore required to be the same width that the other tracks that are leads the same amount of power. Therefore the track width of $7.4v_{BS}$ and $7.4v_{AFS}$ was changed to $1.4mm$. The difference can be seen on the right side of the board, to and from the switch (marked SW1) in figure 4.22, where V2 is to the left and V3 to the right.

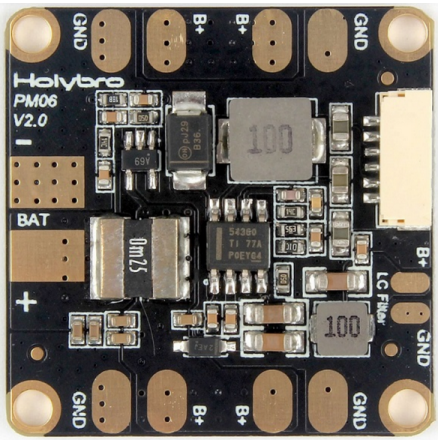
4.7 Ordering and manufacturing of parts

This project is the sixth generation of UiA's drone project. Therefore there will be areas that this project does not focus on, since previous generations have done elaborate work on those areas and it will focus more on the problem description. This information is given to highlight what kind of inventory we had at the beginning of the project and what objects were needed to order, (Appendix H.1), or manufacture.

4.7.1 Reuse of hardware

The RPi CM4, battery, PDU, ESCs, and motors will be the same exact ones used in the previous generation. Because they will be dismantled from one of the drones from last year and be put into the new drone body. For more information about these components see section 6.1 in the last year's project, [79].

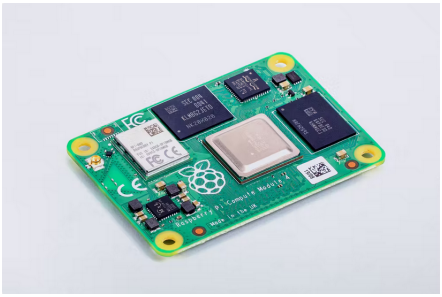
(a) Battery



(b) Power distribution unit

(c) Electronic Speed Controller

(d) Motor



(e) RPi CM4, [69]

Figure 4.24: Picture of components reused from the last year's project [79].

4.7.2 Mounting components on PCB

The components were attached to the PCB using soldering paste, a pick and place machine, and then to heat them up to harden the paste. The process where you heat up the soldering paste to have it flow to the right contact pad is called a reflow process. It is currently the primary process for attaching components to a PCB. Before placing the components on the PCB it is deposited some solder paste on the component pads for then to place the component on the solder-filled pad, next the PCB is normally placed in a reflow oven where the PCB is heated up to match the heating profile necessary. For the paste used in this project, the heating profile is illustrated in (Figure 4.25). During the heating process, the paste is turned into liquid before turning into a solid alloy. [130]

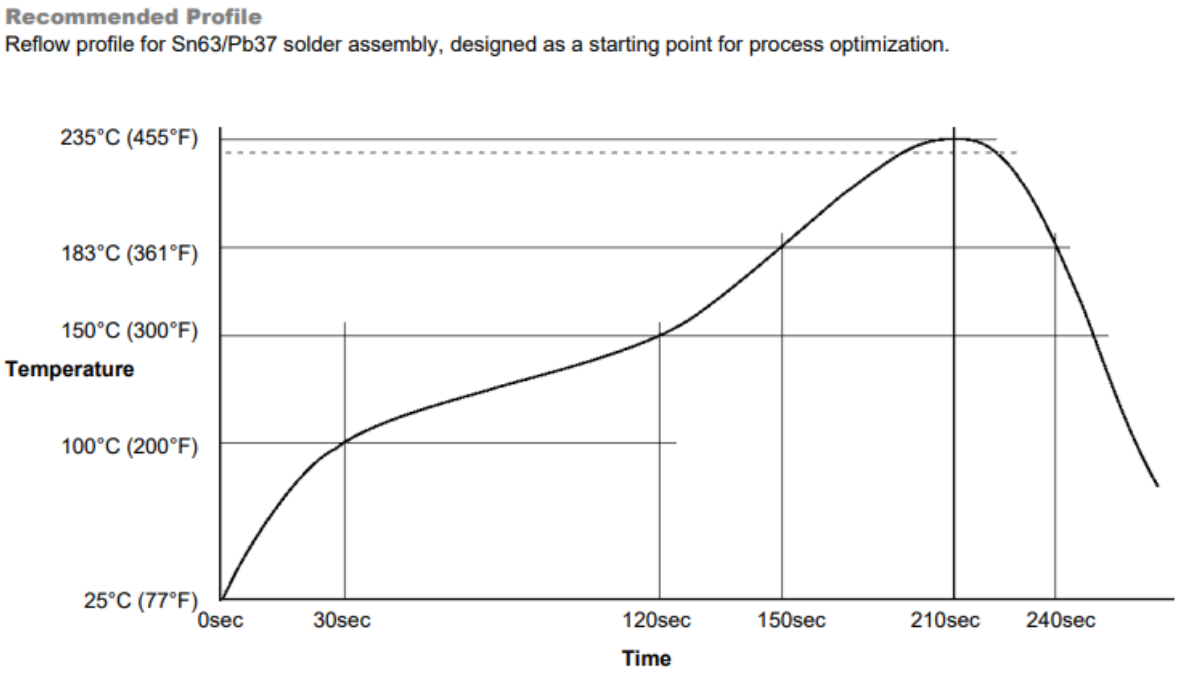


Figure 4.25: Reflow profile for the solder paste [31]

4.7.3 HIROSE connector

The HIROSE connector in figure 4.32 is used between the joint of the RPi CM4 and the PCB (Section 4.3). It transfers information and power between the two. The HIROSE connector in this case has 100 pins and we use two of them that make up 200 different connection pads, which makes the soldering pads small and difficult to attach properly (Section 6.2.2). Since the PCB is a continuation of last year's work [79], it was not chosen specifically by this year's project group, but it was decided that they were the best solution for both the connection, but also for the firmness and how stable the CM4 is when connected to the PCB.

After the attachment of the connector to the board, it was necessary to take it under the microscope to check for any short-circuiting or if the connector was fastened correctly. This was necessary to see if the small solder pads had lined up perfectly.

4.7.4 Evolution of PCB component mounting

Figures 4.26, 4.27, 4.28, 4.29, and 4.30 follows the process of attaching the components, as pictured in said figures they also show its progress and the skills developed per PCB.

Figure 4.26 shows the PCB before the soldering and attachment of the components.

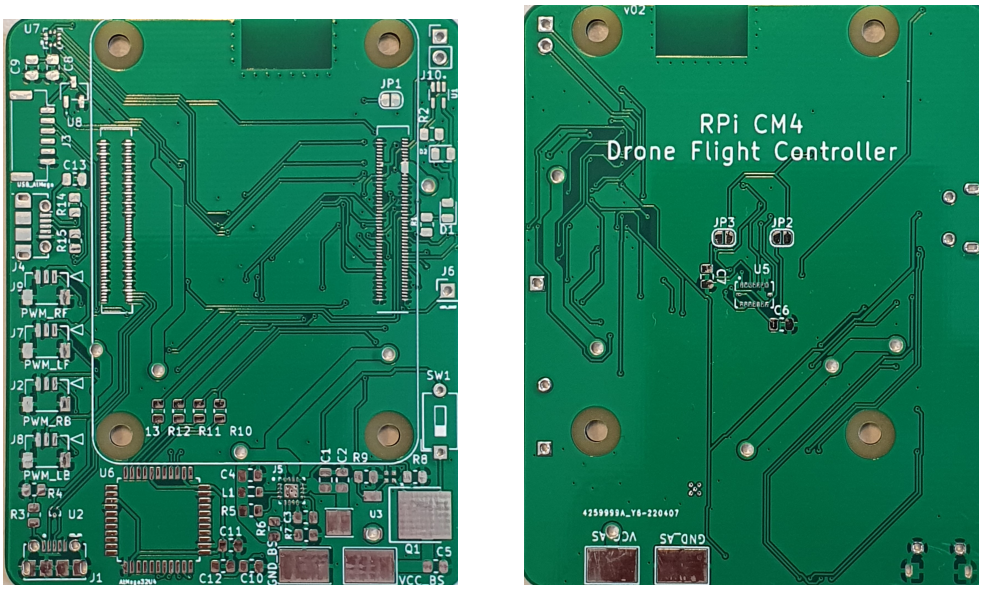
Figure 4.27 shows the first attempt with one HIROSE connector and the essensial component to power the Raspberry Pi CM4.

Figure 4.28 shows the PCB with everything neceserry to power th RPi and two HIROSE connectors.

Figure 4.29 shows the PCB With everything necessary to power the RPi, and the white wire to get 3.3V to the IMU attached on the backside of the card without the LDO, as it had not arrived yet. The voltage regulator (J5) was malfunctioning on this iteration, which led to the red wire on the backside of the PCB which gives 5V directly from the power supply.

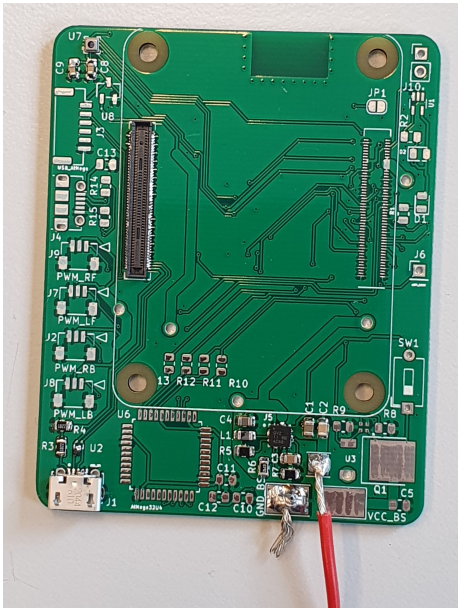
Figure 4.30 is a continuation with also operating LEDs and an LDO to give 3.3V to both the barometer and the IMU, and everything for the micro-USB to work.

Figure 4.31 is the final iteration with every intended component mounted.

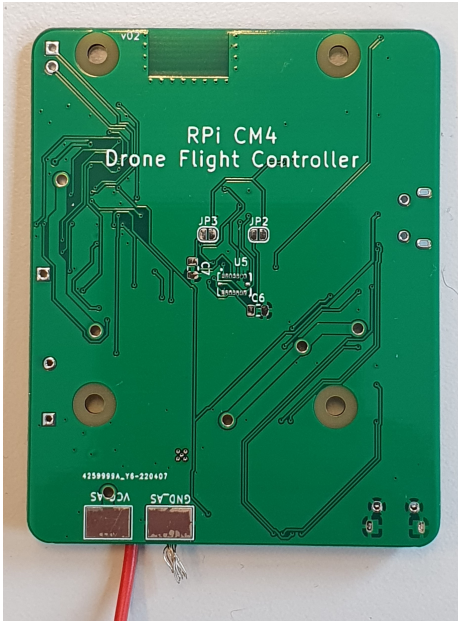


(a) Front (b) Back

Figure 4.26: Empty PCB

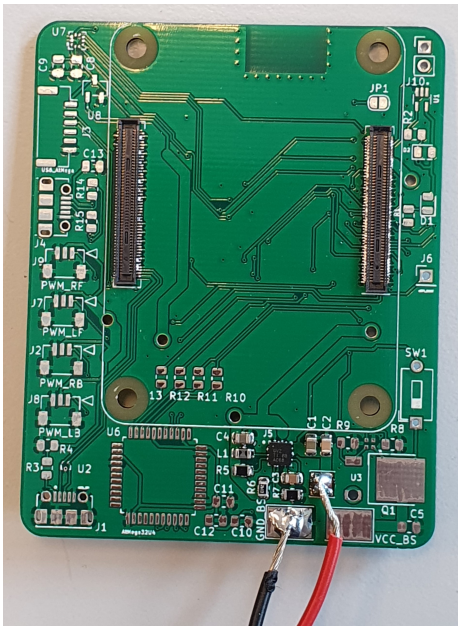


(a) Front

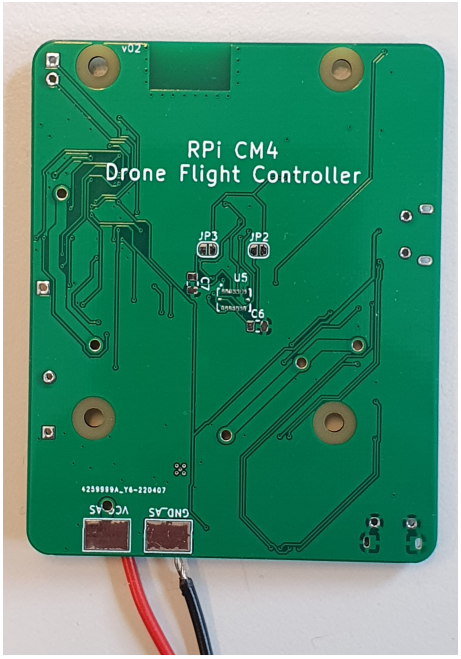


(b) Back

Figure 4.27: First iteration with attaching the components

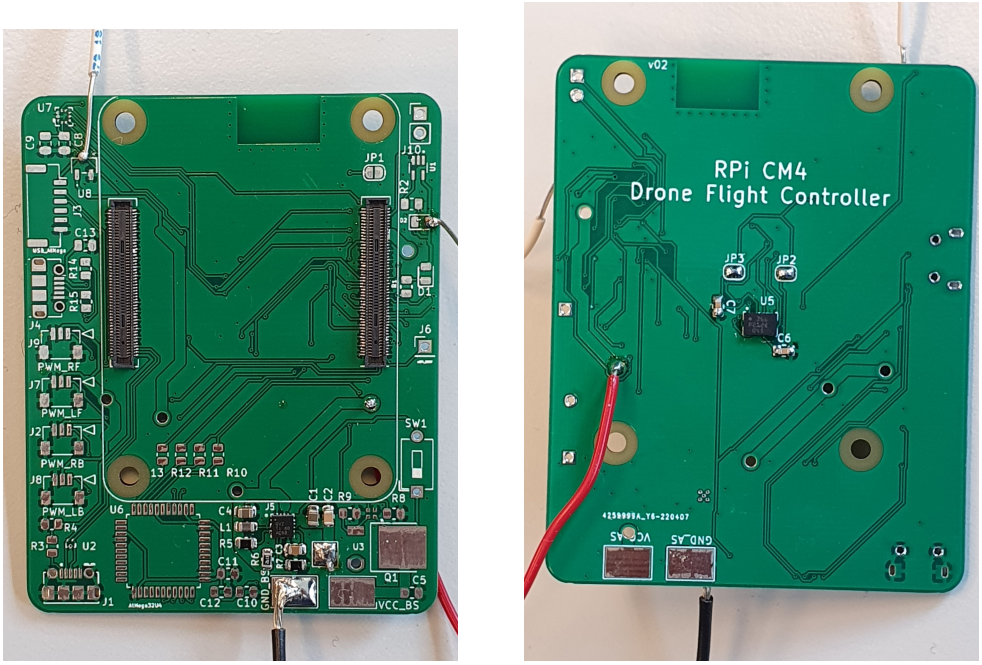


(a) Front



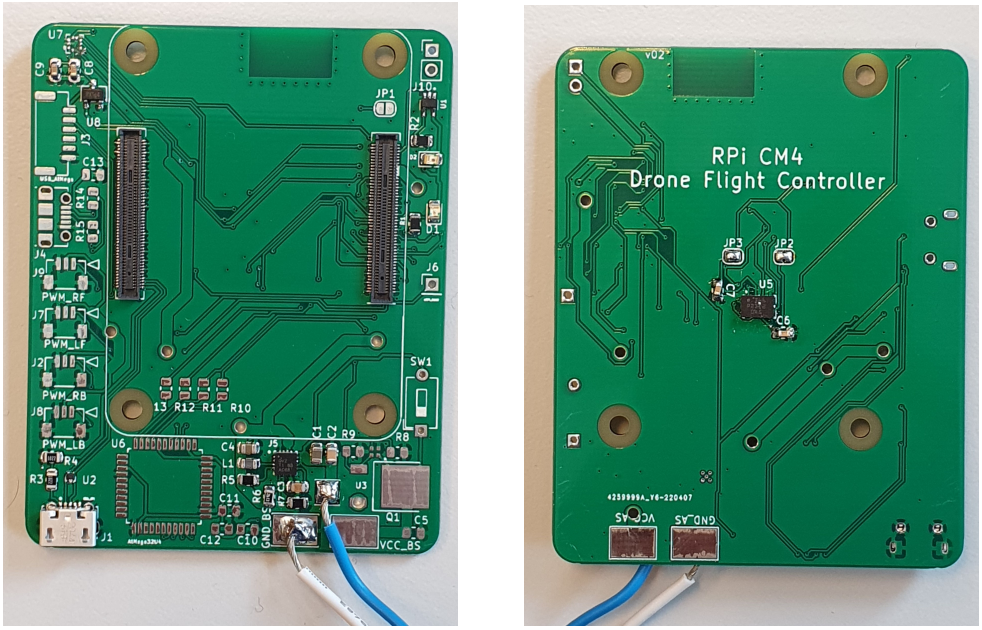
(b) Back

Figure 4.28: Second iteration



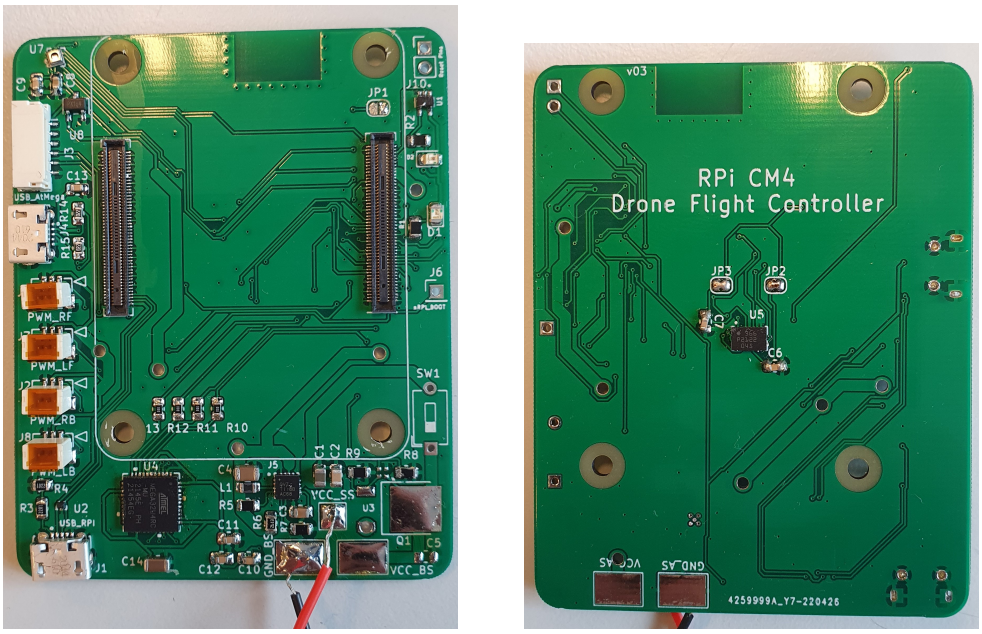
(a) Front (b) Back

Figure 4.29: Third iteration of the solder board



(a) Front (b) Back

Figure 4.30: Forth iteration of the solder board



(a) Front (b) Back

Figure 4.31: A completed PCB with every component needed.

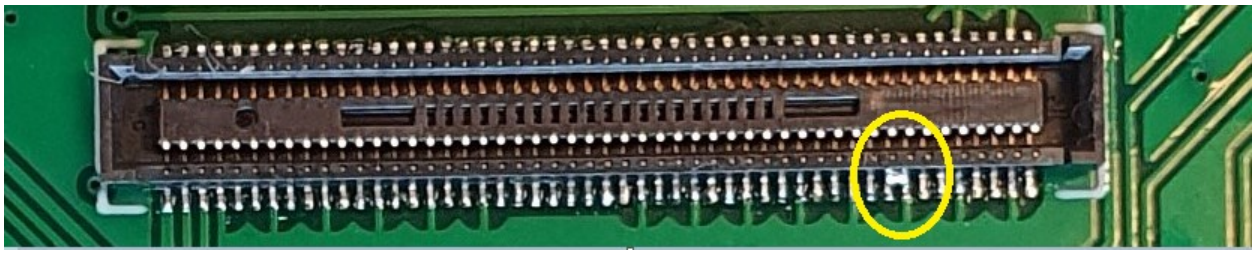


Figure 4.32: HIROSE connector on the final PCB, with a yellow circle to show a short circuit on the connector

4.7.5 Soldering stencil

The soldering stencil (Figure 4.33) was ordered alongside the PCB, which means that it was pre-cut to fit perfectly over the PCB. This allowed the dispensing of the solder paste to be much more precise. Instead of applying the paste under a microscope to be able to hit each solder pad with the right amount of paste. It was now possible to apply the paste over the stencil and softly smear it out.

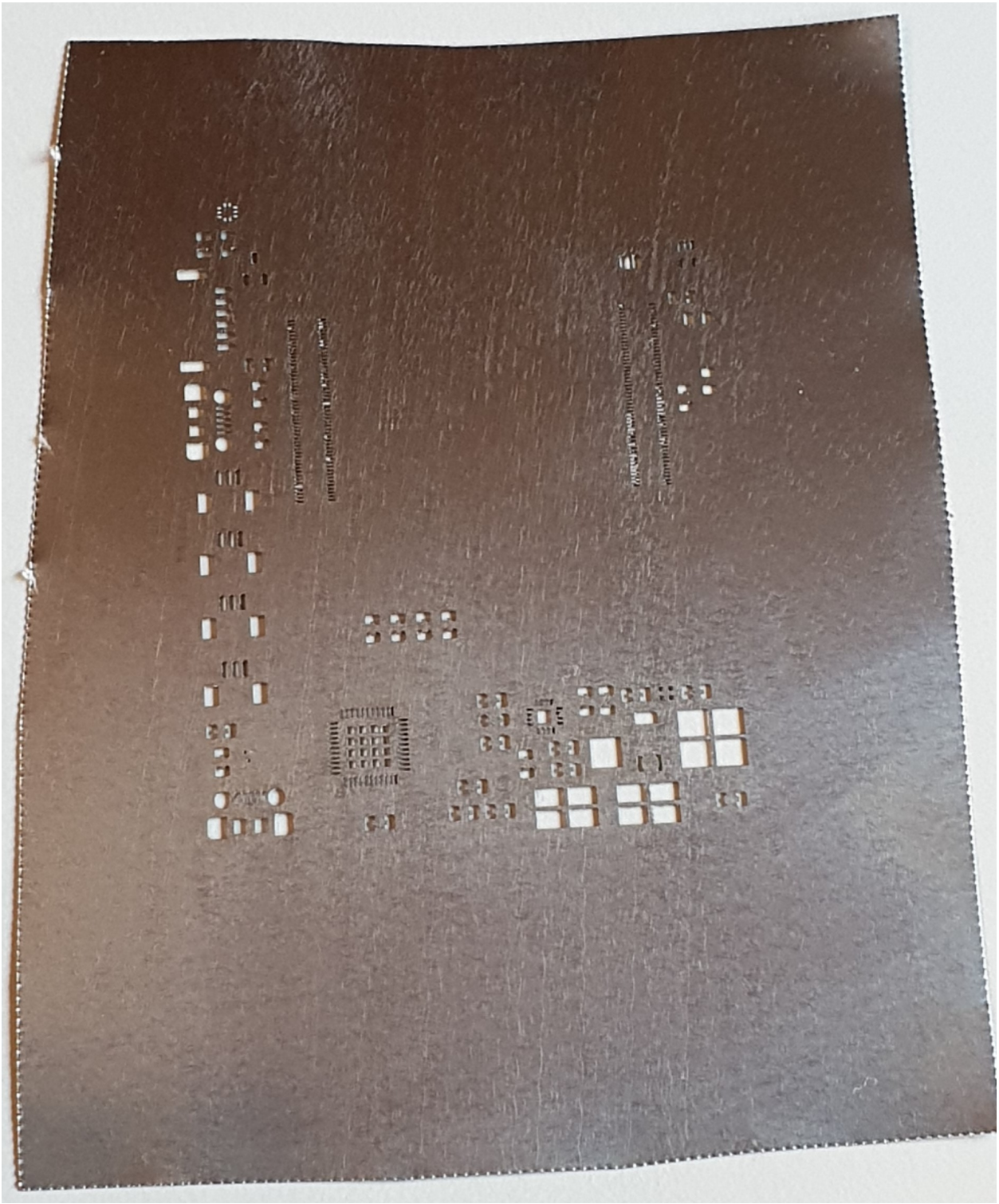


Figure 4.33: Soldering stencil ordered for the last iteration (Figure 4.31)

4.7.6 Use of Time of Flight sensor

For the drone to orient itself among its surroundings an object detection system is needed. ToF sensor provides a view of its environment without colliding with the rules regarding cameras on drones from Luftfartstilsynet [78]. The VL53L5CX sensor was the sensor of choice in this project, due to its great field of view, figure 4.34. In addition to this UiA already had multiple sensors in stock at the time of this project. Which reduces cost, time, and eliminates any possible supply constraint. The ToF sensor is mounted to a shuttle board from Sparkfun, [115]. Its big size and high price make it attractable to develop an own shuttle board to reduce both of these aspects, due to other priorities, this was not further looked at.

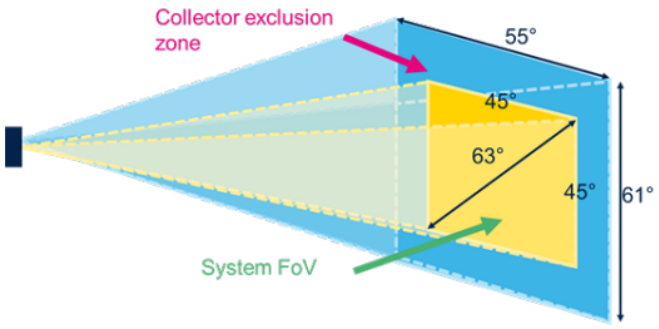
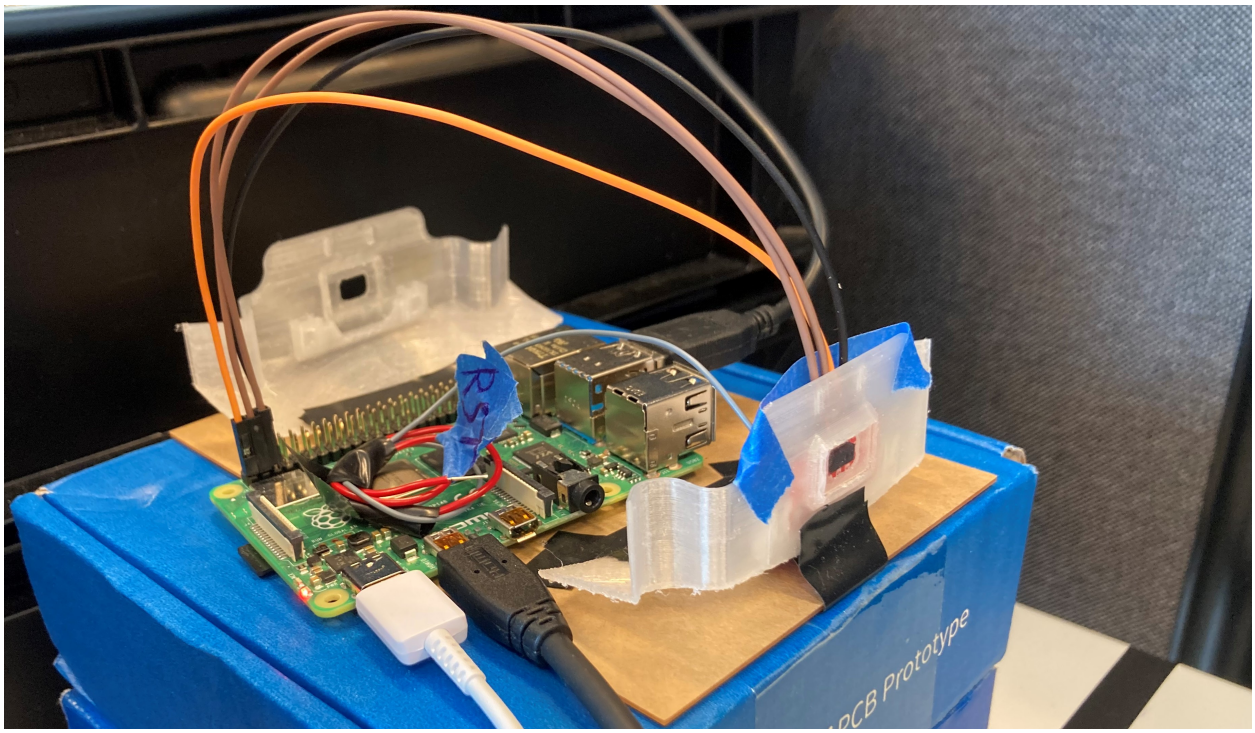


Figure 4.34: Field of View for the VL53L5CX, [119].

To understand how an acrylic sheet affects the ToF sensor's performance, a test will be performed. There will be taken a control test where there will be not mounted acrylic in front of the sensor. For reason of comparing, there will also be one test where an acrylic sheet will be placed in front of the ToF sensor. The data from these tests will be compared in order to see how big the deviation is. Both tests will be taken at the test station in figure (b) below, 4.35.



(a) Close up of the sensor and Arduino

(b) The target area, one meter between the two black lines

Figure 4.35: Station constructed to do tests with ToF sensor.

4.8 Development of structural parts

All of the 3D printed parts used in this project, except for the first three iterations of the Arduino Nano (Section 4.8.3). Started as parts developed by a parallel bachelor project, *Drone Arena for Reinforcement Learning*, [61]. However, every part used was altered to fit the components in this project.

4.8.1 Fitting the ToF-sensor

By using the updated drone shell configured by a parallel project, [61], as a platform for developing a fitting for the time of flight sensor. In order to fit the sensor into the drone, a tight slot was created on the end wall between the two anchor points for the battery holder. What locks the sensor in place is the mid-plate that holds the PCB, this could be seen in figure 4.37. What differentiates the first and second iteration of the slot, is that the latest have the slot closer to the end wall, in order to make a tighter space between the acrylic and ToF sensor. There was no need for a thicker wall, so it was slimmed down and made a fillet. This helped with both weight reduction and made it easier to print, for the 3D-printer. As could be seen from figure 4.36 below, a cutout is made in the second iteration this is to accommodate a soldered wire for resetting the sensor.

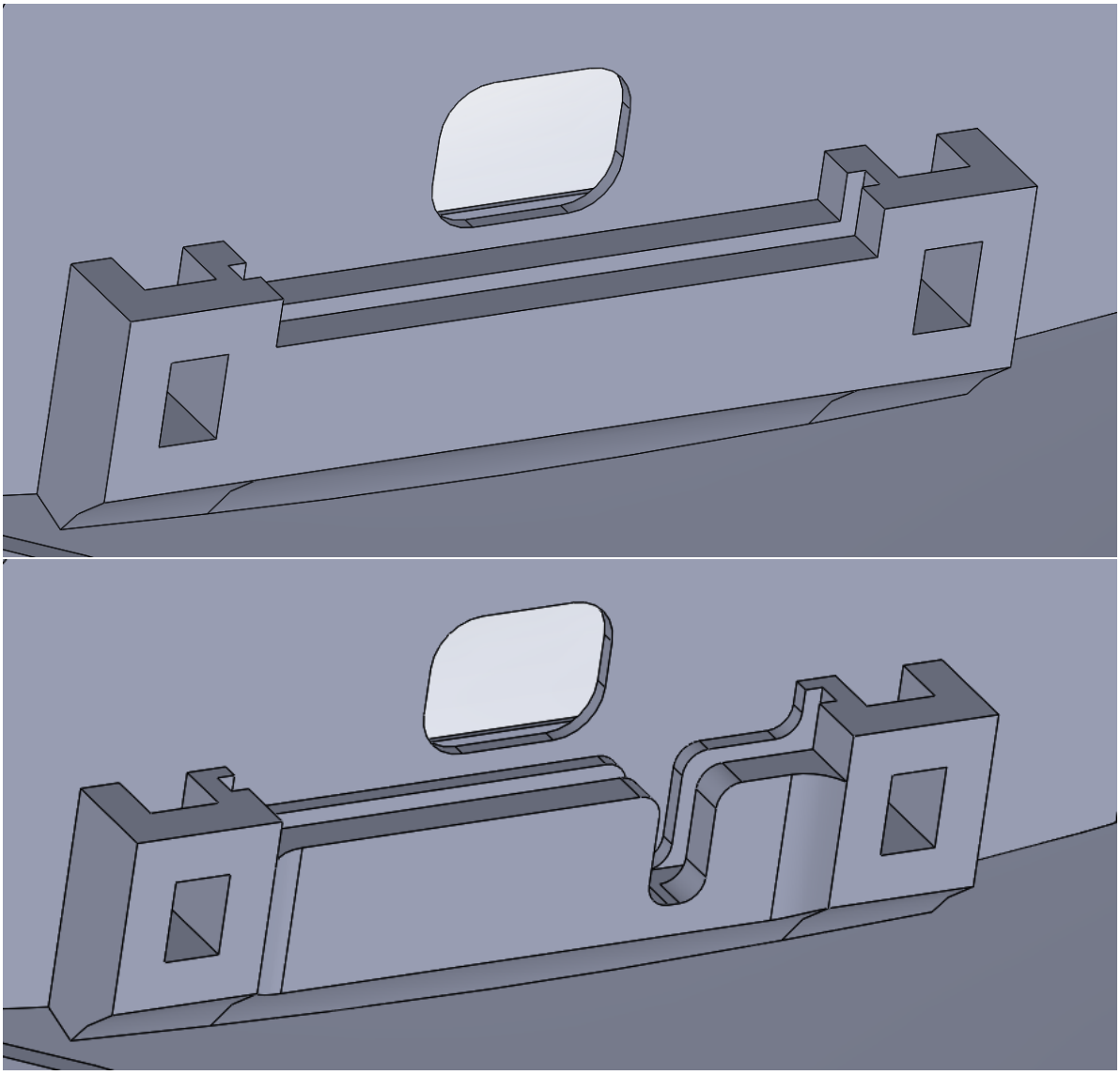


Figure 4.36: Top figure is the first iteration, and the bottom figure is the second and final iteration of the slot for the Sparkfun ToF sensor.

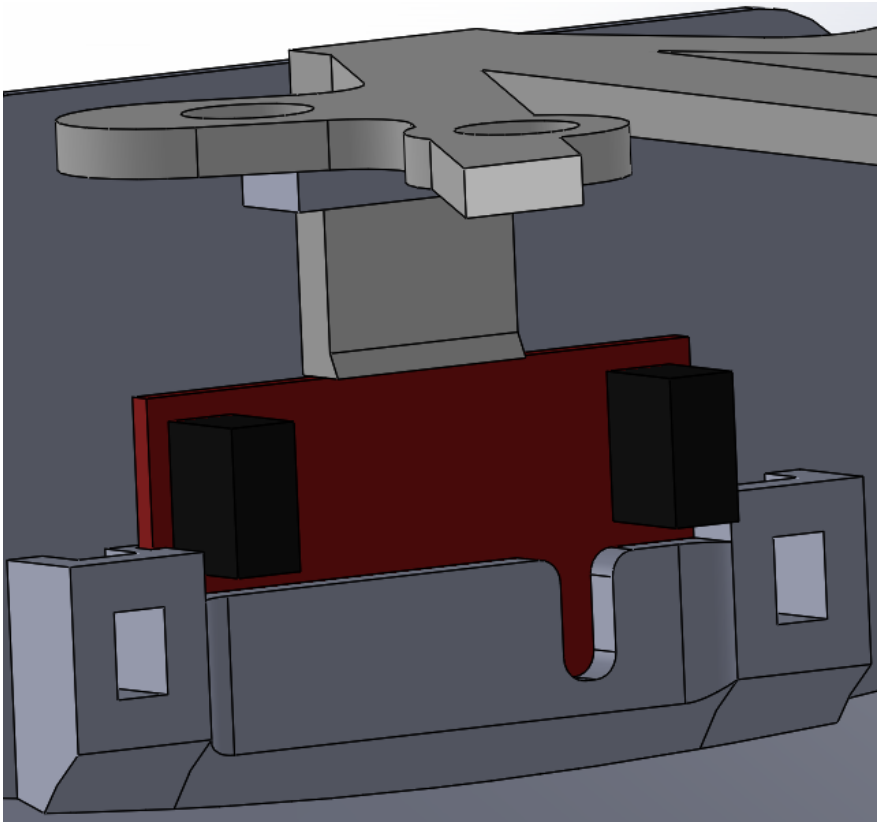


Figure 4.37: Demonstration of how the Sparkfun ToF sensor is mounted in the drone

4.8.2 Protective lens

The ToF-sensor is very sensitive to debris if exposed to the elements. In order to combat this, a protective element is needed. This problem is solved with a 1mm thick acrylic sheet put in front of the sensor, a description of the acrylic used see section 2.2.2. To keep the sheet in place, extrusion from the drone body is made (Figure 4.38). The second iteration is double the thickness of the first iteration, due to the first one being too shallow, too little overlay over the acrylic, and the 3D printer struggling with printing such detailed and small features. The dimensions of the overlay frame, which keeps the acrylic from popping out, are made 1mm smaller both in width and height in the second iteration.

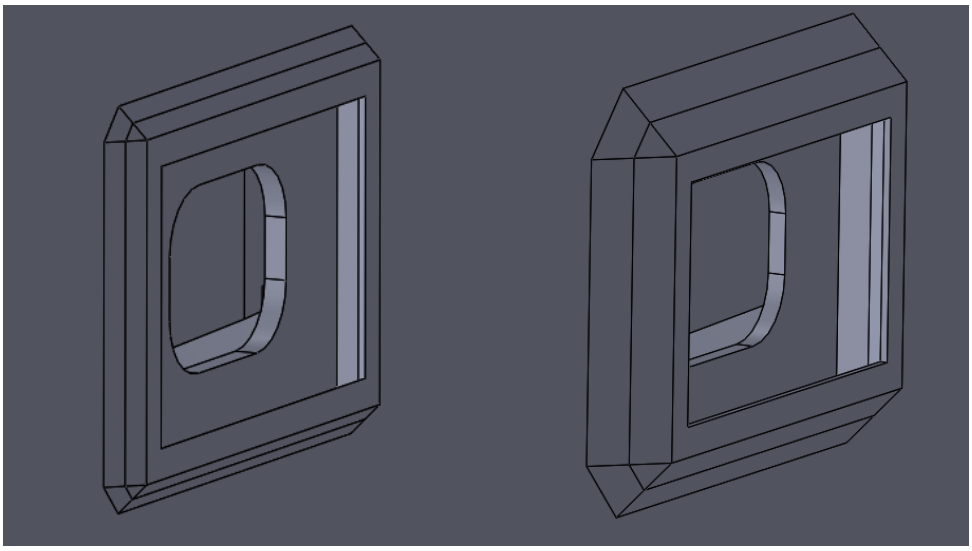


Figure 4.38: Left is the first iteration, and right is the second and final iteration of the lens holder.

To cut the acrylic sheet into the wanted dimensions, 10mm∗15mm, a laser cutter is to be used. The reason to use a laser cutter is that it leaves a cleaner edge and has the possibility to cut out multiple lenses in one process. This makes substantial time savings if producing multiple drones.

4.8.3 Mounting the Arduino Nano

The Arduino Nano is going to be fitted into the drone body. As a replacement for the ATmega32U4, due to having the wrong footprint in V2 of the PCB, see section 4.6.1. The reason for this implementation is to have a continuation of the software development, and not wait for the arrival of the V3 PCB. For the first three iterations of the brackets, the board was going to be mounted above the Raspberry Pi CM4. For the fourth iteration, a total redesign and placement were developed. The reason for this was that the previous iteration was building too much height, this would have made serious complications for the drone body, and therefore added unnecessary weight. The new placement is alongside the battery, this makes could potentially make the weight distribution uneven. But the weight of the Nano is just 7g, and most of the wires will have to be put on the opposite side of the battery. This makes the potential uneven weight distribution from the Nano negligible.

Iterations 1, 2, and 3

Taking the dimension from the datasheet to Arduino Nano [6], and the dimension of the PCB (Section 4.4.9). By knowing this information it was possible to find the correct spacing needed to place the Nano, in the center of the drone. The first iteration, 4.39, used the holes in the CM4 to make the brackets stay in place, while the second and third iterations were going to be placed on top of the screws used to fasten the CM4 down to the PCB. It came apparent that additional spacing was needed in order to place the Arduino above the CM4, that is the main difference between iterations one and two. Iteration three was developed to be printed in a different orientation. By printing the poles that holds the Nano, flat to the ground they become less prone to fracture. An additional positive note is that there is a reduced need for support during printing in that configuration.

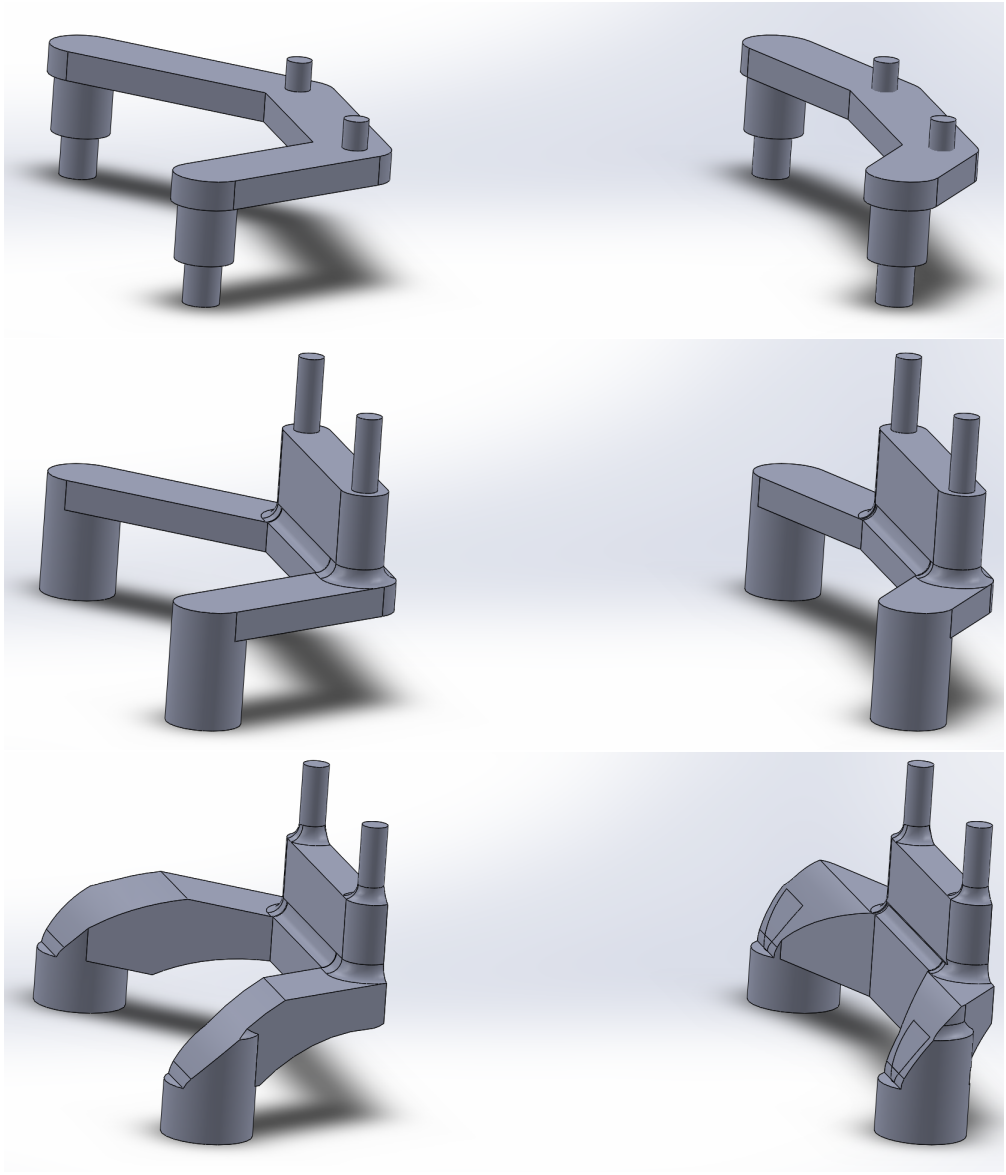


Figure 4.39: Mounting brackets V1, V2, and V3 for the Arduino Nano in chronological order

New placement

After seeing that the 3rd iteration made the Arduino reach too high, the lid for the drone would have to be redesigned. It then came apparent to use some of the space in the drone instead of redesigning. So for the fourth iteration, the Nano was placed alongside the battery. An additional benefit of the new placement is the improved thermodynamics, due to the Arduino being far away from the CM4 processor. By altering the PCB mounting bracket made by a parallel project, [61], and making two extrusions, see the top illustration in figure 4.40. In the tight fit between the lowest pin and the bracket itself, the printer added support. This was cumbersome to remove without fracturing the pin. So in the fifth iteration, this problem was solved by altering the holes in the bracket. This made the support easier to remove since the spacing was bigger. In the fifth iteration, the pins on the extrusion were made shorter, due to the Arduino not needing all the provided spacing. The shortening of the pins gives the Arduino better clearing from other parts when fitting the bracket into the drone.

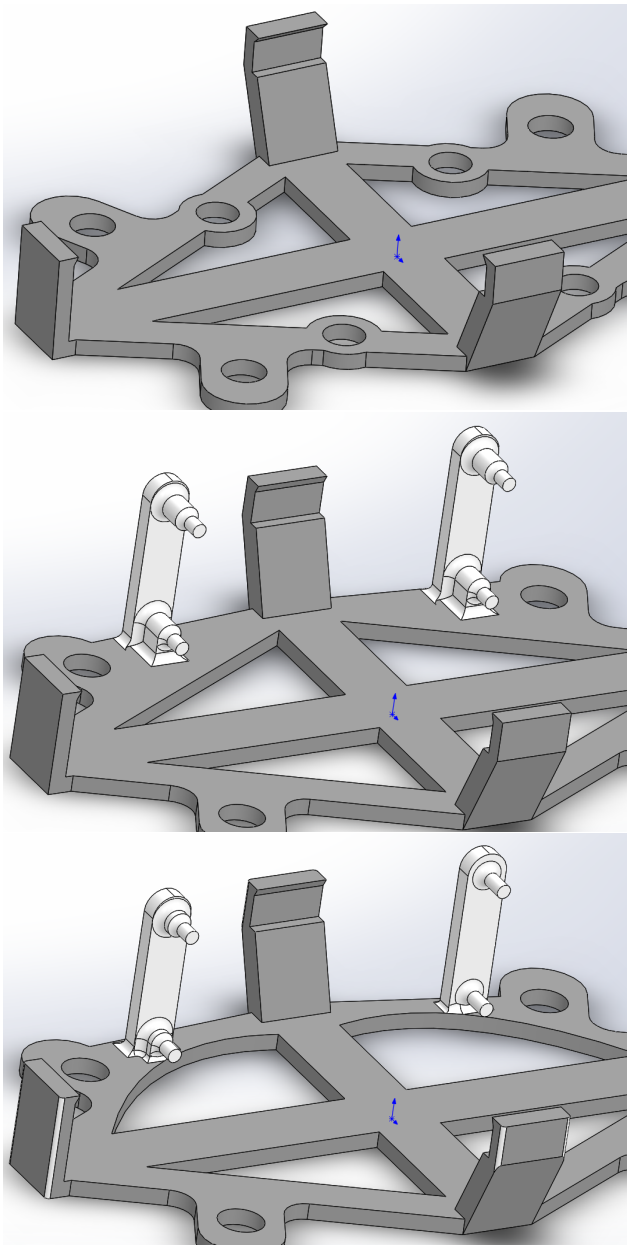
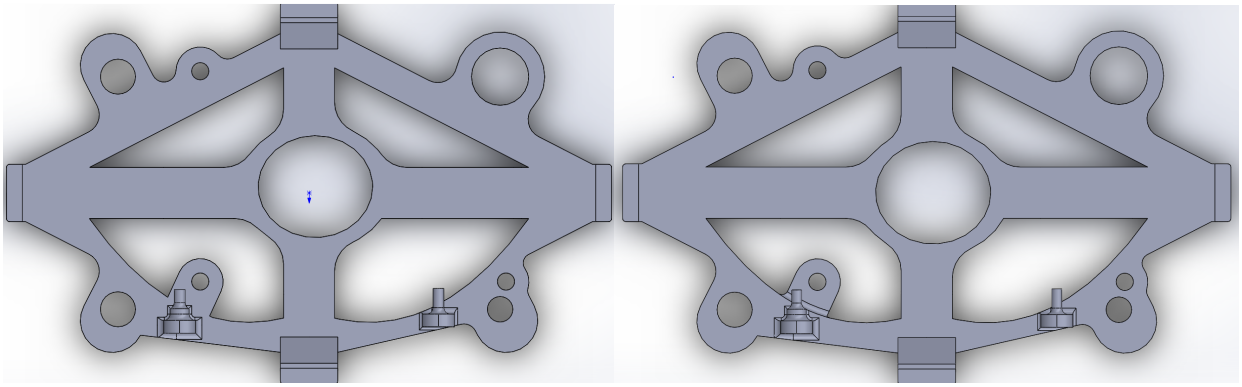


Figure 4.40: The parallel project's original design and 4th and 5th iteration of mount for the Arduino Nano

4.8.4 PCB-mounting bracket

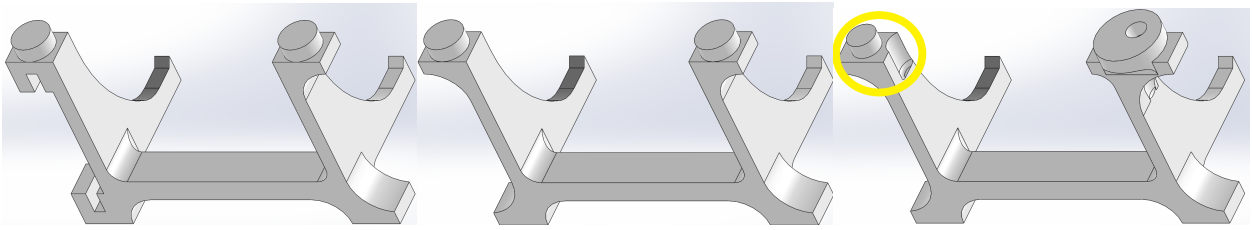
Using the fifth iteration of the mount for the Arduino base for further development of the PCB mounting bracket. The only difference is that it is mirrored, due to misconception of the orientation during earlier design stages. Used the dimension from the PCB, figure 4.14, to put the IMU in the center of the drone. Since the IMU is on the backside, a hole in the bracket was made in order to fit both the sensor and the belonging components. Minimal changes were between iterations one and two, figure 4.41. The difference is moving the hole for the IMU 1mm down to the center, making the extrusion for the bottom left screw mounting 1mm thinner, for better spacing between the bolt and the Arduino. The last change was moving the two left holes for the screws 0.5mm to the right in order to align better with the holes from the PCB/CM4.



(a) First iteration (b) Second iteration

Figure 4.41: Bottom view of the PCB mounting bracket.

The top left hole in the figure above has a bigger measurement than the rest. This is to accommodate a hole from the PCB, within the already existing hole. In order to fasten the PCB to a structural part, the battery holders had to be altered. The original version of this pic the one on the left in figure 4.42 below. The left extrusion on the north version, marked with a yellow ring, has also been altered. This has been changed to a smaller dimension in order to make space for the bolt that is going to the PCB. This mentioned bolt also needs a nut to screw onto, in order to fit the nut the a small portion of the wall was cut, this is also true for the hole on the right extrusion.



(a) Original (b) South version (c) North version

Figure 4.42: The left figure is the file received, the other two are made to fit the needs of this project.

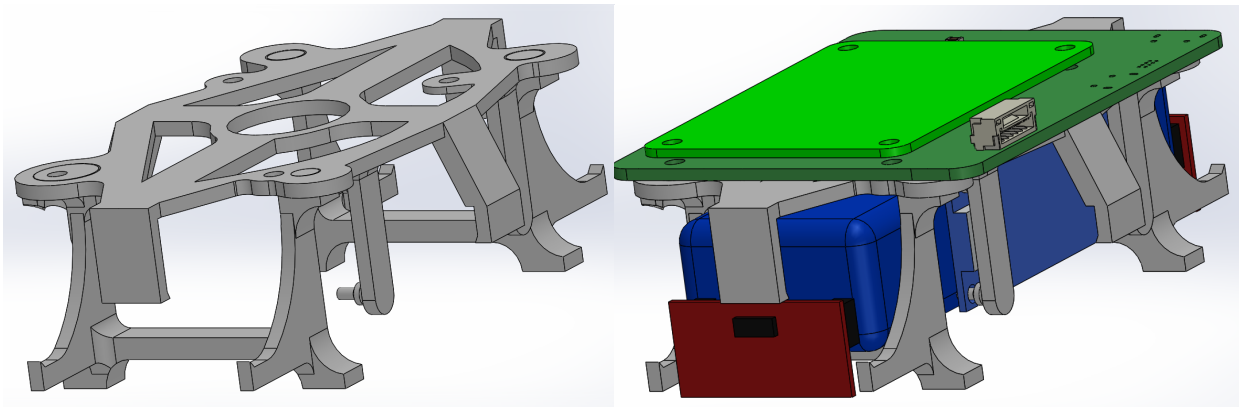
4.8.5 Altercations from 1st to 2nd drone body

The files that were received from a parallel project, [61], were still under development. This created altercations when assembling the internal components into the drone body. The drone shell/body was 2mm thick in V1, but 1mm thick in V2. This meant that the PCB mounting bracket had the be extended by 2mm, without changing the placement of the PCB and CM4. Both north and south battery holders had to be elongated 1mm each in order to fit the updated

bracket. The surface that the motors mount onto had to be extruded 1mm, to help stiffen up the mounts. The brim that connects/locks the bottom part to the top part had been completely re-imagined for version 2 of the drone shell. Copied the sketch from the parallel project, [61], and drew a 3D sketch to make it follow the outer rim of the bottom part.

4.8.6 Internal setup

An assembly was made to see how the different parts would fit together. Every part in the internal setup was originally developed by a parallel project, [61], but was heavily altered in this project in order to fit the components used in this version of the drone.



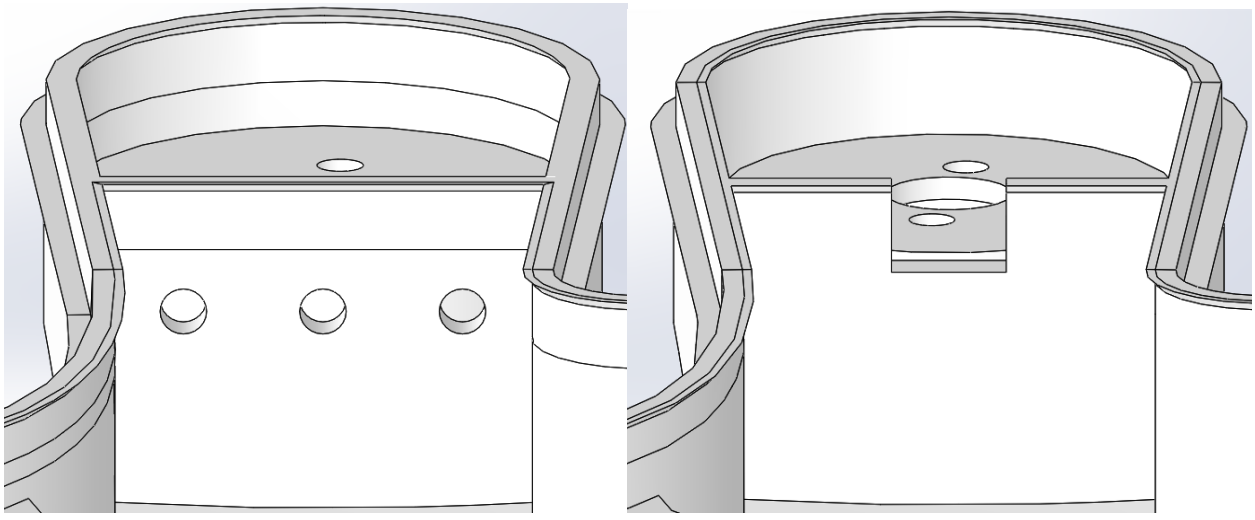
(a) Internal setup without components

(b) Internal setup with components

Figure 4.43: Final iteration of internal layout.

4.8.7 Other changes

During the development of this project, a parallel project, [61], made integrated footings in the drone body for the battery holder, to add robustness. Those got implemented into the final iteration of the drone body.
In order to save time and reduce workload, the wall that separates the motor from the interior of the drone was altered, see figure 4.44. This was to not having cut the wires going the motor and splicing them together again.



(a) Original version

(b) Altered version

Figure 4.44: Changing the motor wall for easier assembly of motors with associated wires.

4.8.8 Production of parts

All of the mentioned parts in this section, PCB mounting bracket, battery holder (north and south), and hull (top and bottom), were 3D printed with UiA's printers. The printers were widely available and easy to use. This meant that rapid prototyping was possible. By doing it this way, it was easy to put away iterations after finding out their limitations in the physical world. All the printed parts in the interior setup were printed in a material called PLA, Polylactic acid, which is very common to use in 3D-printed. However for the hull, a material called PP film, Polypropylene, this was due to having interconnections with a parallel project, [61], and requirements for its project.

4.8.9 Mounting of parts

The motors, PDU, and the PCB stack are the only parts that need tools to be mounted. The rest can be mounted by hand. Due to the PP film used, it is possible to squeeze the acrylic piece in place, even if the lens holder has a smaller size than the acrylic. For step by step tutorial on the mounting process see Appendix J.

4.9 Real-Time Operating System

Up until this point, regular distribution of Linux has been used as the operating system. A regular distribution of Linux is, along with Windows and macOS, a general-purpose operating system. When operating with embedded applications, there are deadlines that must be met with good accuracy. Although a GPOS also has deadlines, they are generally not as reliable as they can be with the use of a real-time operating system. Since the drone needs to respond fast to changes from for example wind and positional error, it is of high interest to use an RTOS for this application. There are many RTOSs out there available as open-source and publicly accessible systems. Previously both NuttX and a real-time Linux patch have been used as RTOS for the UiA drones.

4.9.1 RTOS Comparison

NuttX and ChibiOS are as mentioned two of the most popular RTOS distributions used for drones. Flight firmware such as Ardupilot and Pixhawk utilizes these operating systems in their applications [136]. The initial thought was to try and implement one of these operating systems in the drone, to then be able to use for example Mission Planner to calibrate and fly the drone [8]. The use of NuttX was quickly realized not to be possible, as according to the NuttX platform support manual, the support for the Raspberry Pi platform is incomplete [91]. It is also based on the BCM2835 processor, and the CM4 we use has the new BMC2711 processor. It was also only documented for use on the Raspberry Pi Zero, which this project does not use.

ChibiOS on the other hand has been used on the Raspberry Pi platform. The fork from Steve Bates on GitHub was reviewed for our drone application [19]. The repository was cloned and investigated, and it was also attempted to implement it on the RPi. However, nothing worked. After the review, it was decided this would not suffice for our application. The repository is 10 years old and is also a work in progress. Additionally, there was no documentation on the ChibiOS source site about the OS on a Raspberry Pi [44]. Drivers for SPI and I2C are not fully developed, and since SPI is the main communication interface used on the PCB the OS was deemed not optional.

As this part of the project uses a Raspberry Pi with the Raspbian OS, which is a Linux distribution, the real-time patch for the Linux kernel is a good candidate and an easily obtainable option with good real-time capabilities [4].

The patch converts the kernel to a fully preemptible kernel and implements priority inheritance. This is done using spinlocks, which make the thread wait in a loop until a lock is required [86]. This means that the kernel will prioritize tasks based on their priority setting, which can be set by the user. This enables the user to define what tasks are more important than others. Additionally, interrupts are now converted into preemptible kernel threads [4]. This means that if a sensor sends an interrupt, the kernel converts this to a thread, which is the highest level of code executed by the processor [131]. An interrupt is usually sent to signal a task that requires immediate attention, and converting this to an active thread in the kernel instead gives the user the option to decide what to do with it [127]. This thread can then be assigned to a priority by the user, and be executed when wanted. This makes the task assignment easier for the user and makes the system react according to the implementation set by the user. For example, the PID control of the system needs to always be running, and the data needs to be prioritized to ensure that the motors receive the correct signals. By assigning a high priority to a PID thread, the operating system knows that this is to be executed first above for example a new interrupt from the IMU.

Based on this comparison, the Linux Preempt patch was selected to be the RTOS of the system. A guide on how to patch the Linux kernel is located in Appendix A. This gives a detailed walk-through on how to download, patch, and configure the Linux kernel to achieve real-time capabilities.

4.9.2 Test of the Real-time Linux kernel

To test the real-time kernel, the RT-tests from the Linux Foundation were used [49]. By comparing the real-time kernel to the regular kernel, one can see if the real-time changes have a big impact or not and how the real-time kernel handles for example latency and signal transmitting. Please be careful when reading the plots, as the y-axis differentiates in some of them while still being the same test.

Beginning with downloading the test suite to the Pi and unzipping the folder. The folder contains 14 tests. Reading the README file, it is shown how some of the programs work. First, we make sure that the build-essential and libnuma-dev packages are installed on the Pi. These can be installed using the apt-get install command. 5 of these tests were chosen, as they give an all-around perspective of the real-time capabilities of the kernel. The test guides are acquired from the Arch Linux website [128], where the developers of the tests explain in detail what the tests measure. There is also a test involving a mathematical problem, described by LeMaRiva when testing the 4.19-rt kernel [76].

Cyclictest

The first test conducted was the cyclictest [51]. This is a popular test for real-time systems as it analyses the worst-case latencies of the system for both hardware and operating system. This gives a good indication of the performance of the system, and is an excellent comparison between the real-time and regular kernel. Executing

```
pi@raspberrypi:~/rt-tests/ sudo ./cyclictest -a -t -n -p99 -his400
```

runs the test with one thread per CPU core with priority 99 which is the highest priority. This parameter comes from the pthread library, which is why this library is vital when setting priorities. This then gives us the performance of each of the four cores on the RPi. The his400 command plots a histogram of the results that are 400 μs or below, as shown in figure 5.13. This test should ideally be run for a long time, that is to capture the latency when the CPU is under load and stress. This test could be run again when the drone is under flight to capture this result, as conducting a full test will take over 5 hours. Either way one can see that the real-time system outperforms the regular system, although not by much. This number might change as the load time on the CPU becomes longer, but the goal here is to get a system that performs faster than the regular system, and this real-time patch is 14 μs faster. Also to be noted, these are the maximum latency's! Below the graphs in figure 5.13 there are printed the minimum, average, and maximum latency's for the four CPU cores. As one can see here as well, the latency's are significantly lower in the real-time kernel across all categories.

Ptsematest

This test starts with two threads and then measures the latency of the interprocess communication with POSIX mutex [45]. As explained in the guide, the program synchronizes the two threads with pthread_mutex_unlock and pthread_mutex_lock. It then measures the latency between sending and getting the lock. To run the test, execute:

```
pi@raspberrypi:~/rt-tests/ sudo ./ptsematest -a -t -p99 -i100
```

This starts the test on the current processor, with the default number of threads, which is 1 if not specified. It also sets the highest priority for the process, which was previously explained is 99, and the number of iterations. The test duration can also be set by using -d. This test then runs on all 4 CPU cores, where CPU 0 receives the highest priority, and CPU 1 gets the second-highest and so forth. The end result is shown in figure 5.15.

Migrate test

The next test is used to check if the task scheduler ensures that the highest priority tasks are executed and running [110]. This is important to check if one sets the highest priority for a task that needs to be met, and if it is not, this could cause a system failure or that for example the drone cant keep leveled because the PID-algorithms run too slow. Run the test by executing the following:

```
pi@raspberrypi:~/rt-tests/ sudo ./rt-migrate-test -l -p
```

Now the result will display how the different tasks were executed in each loop based on priority. Since only -p is set and the priority is not specified, the processor uses the default configuration and starts at priority 2 and works its way up to priority 6. The -l indicates the default number of loops, which is 50. The results are displayed in Tables 5.1 and 5.2.

Sigwaittest

The sigwaittest is a signal-wait test [46]. Two threads are started, and the latency is measured between sending and receiving a signal.

```
pi@raspberrypi:~/rt-tests/ sudo ./sigwaittest -a -t -p99 -i100
```

This, as said before on the other tests, runs the program on the default processor with the default number of threads. The priority is set to the highest, and the test runs for 100 iterations. These results are shown in figure 5.16.

Svsematest

This test is similar to the sigwaittest. It starts with two threads which are synchronized via SYSV semaphores, or system 5 semaphores [47]. Semaphores are used for data accessing during for example multitasking operations, which is important in drone applications. This essentially means that the semaphore can be accessed by any thread in the process, but it can block or unblock the access based on its value. If for example the semaphore is being decremented by a thread and the value is negative, the thread blocks itself, and to be unblocked another thread must increment the semaphore. This is the only way to access a semaphore, as its current value can not be read. More on semaphores can be found in this book [38]. To run the test:

```
pi@raspberrypi:~/rt-tests/ sudo ./svsematest -a -t -p -i100
```

This uses the same parameters as the sigwaittest, and they mean exactly the same. The results can be viewed in figure 5.17.

N-Queens Problem

This test is made, as mentioned, by LeMaRiva [76], which also has a guide to patching the Linux kernel. This test gives the processor a problem to solve, and by measuring the time it takes for the processor to find all solutions, the regular and real-time systems can be compared. The test also includes the CPU and GPU temperature when calculating the solutions. This can show how hot the processor gets during calculation and is also a part of the final plot. The problem is cloned from GitHub from this link [75], and requires python3 packages on the Pi to execute. This can always be installed using apt-get install python3-pip. Run the following command inside the N-Queens-problem to run the program.

```
pi@raspberrypi:~/N-Queens-problem/ python3 queenpool_multithread.py ...
    multithread_output.csv 12 4 10
```

This defines that there are 12 queens to be found, there are 4 threads started, and the test runs for 10 iterations. Results are found here 5.18. The results of these tests are further discussed in the discussion chapter 6.5.1.

4.10 NASA F' Framework

4.10.1 F' Description

F' is described as "*A flight software and embedded systems framework*", [87]. It is a framework developed by NASA at the Jet Propulsion Laboratory in the USA. What a framework is, is explained in the theory chapter 2.3.6. The framework is open-source, which means that it is open to everybody who wishes to use it through NASA's GitHub page. The framework has been used by NASA in various space applications, such as the CubeSats and SmallSats projects, [89]. The framework contains designated fprime-tools, libraries, and test tools. In this evaluation, we will go deeper into the framework and test the applications and libraries, and also try to implement the drone components into the framework.

4.10.2 Initial setup, installation, and testing

To access the F' framework, one has to clone this onto a host computer. In this case, Ubuntu 20.04 LTS is being used as a virtual machine running on Windows 10. Since the framework is open source, we can simply clone the GIT-repository into a dedicated folder on the VM. The repository includes a README and INSTALL.md file, which contains all the requirements for downloading and installing the framework. These are shown in figure 4.45

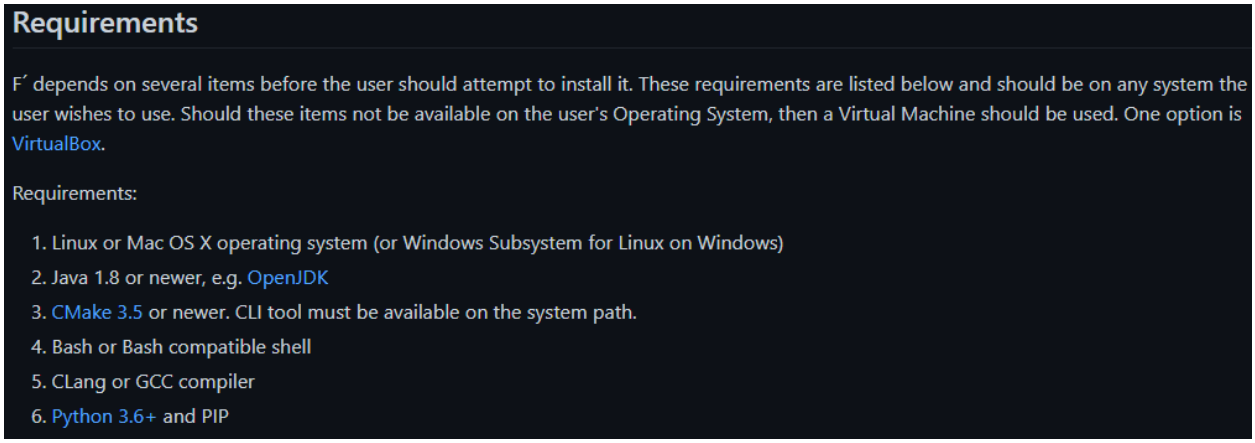


Figure 4.45: System Requirements for F'

All these requirements must be met to run the framework. To check if the system is meeting these requirements, one can run the following commands in the Ubuntu terminal.

Listing 4.1: System Check

```
$ java -version
openjdk version "11.0.13" 2021-10-19
OpenJDK Runtime Environment (build 11.0.13+8-Ubuntu-0ubuntu1.20.04)
OpenJDK 64-Bit Server VM (build 11.0.13+8-Ubuntu-0ubuntu1.20.04, mixed ...
    mode, sharing)

$ cmake --version
cmake version 3.16.3

$ gcc --version
gcc (Ubuntu 9.3.0-17ubuntu1~20.04) 9.3.0

$ g++ --version
g++ (Ubuntu 9.3.0-17ubuntu1~20.04) 9.3.0



$ python3 -V
Python 3.8.10

$ pip3 -V
pip 20.0.2 from /usr/lib/python3/dist-packages/pip
```

If the system does not meet the requirements, one can run the apt-get update and upgrade commands to ensure that the system is up to date. From there the components can be installed by executing these commands in the Ubuntu terminal.

Listing 4.2: Requirements for installation

```
Java:
$ sudo apt install default-jre

Cmake:
Can be accessed from the Ubuntu Software Application

GCC and/or G++ compilers are automatically installed when setting up the VM.

Python 3 and PIP3:
$ sudo apt-get install python3.8 python3-pip
```

Installation and testprogram

When installing F', the INSTALL.md file from the NASA repository should be followed [11]. As mentioned, the repository should be cloned into a suitable folder on the host machine. In this case, it is cloned into the home directory, which is shown in figure 4.46. The F' framework needs some specific python-packages, which can be installed as mentioned in the file by executing:

```
$ sudo pip install --upgrade fprime-tools fprime-gds
```

The sudo command is important here, as you want the tools to be accessible in the fprime folder you cloned the repository to. If sudo is not executed, the tools will be automatically placed in the .local folder which requires you to add the path to the tools when you execute the Autocoder and GDS commands.

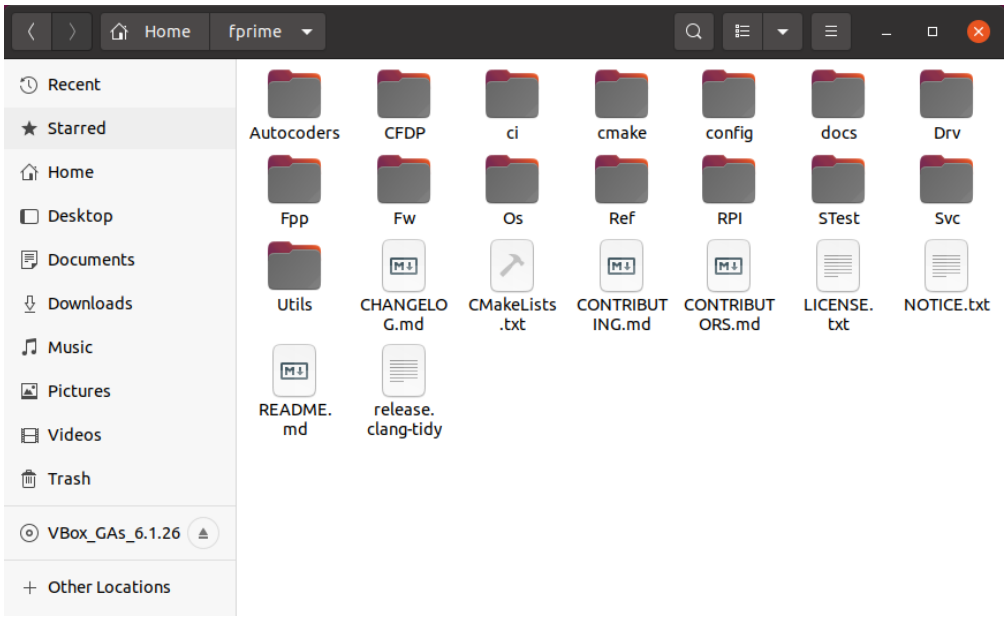


Figure 4.46: F' folder

When this is done, the F' installation is complete. To test the framework, the example program in the Ref folder should be executed. This will test the autocoder and gds-tools that were installed in the previous step. Doing this test will ensure that the installation was successful, and also show the ground station GUI for the framework. Following the steps in the README-file, this is executed in the command line.

```
$ cd fprime/Ref
$ fprime-util generate
```

This generates a build directory called build-artifacts in the Ref folder. To build the application, run:

```
$ fprime-util build --jobs "$(nproc || printf '%s\n' 1)"
```

To then test the Global Distribution system, run the HTML graphical user interface by the given command:

```
$ fprime-gds -g html -r /home/fprime/Ref/build-artifacts
```

This will open up the web browser, and display the F' ground system as seen in figure 4.47. Now one can inspect the different aspects of the ground system.

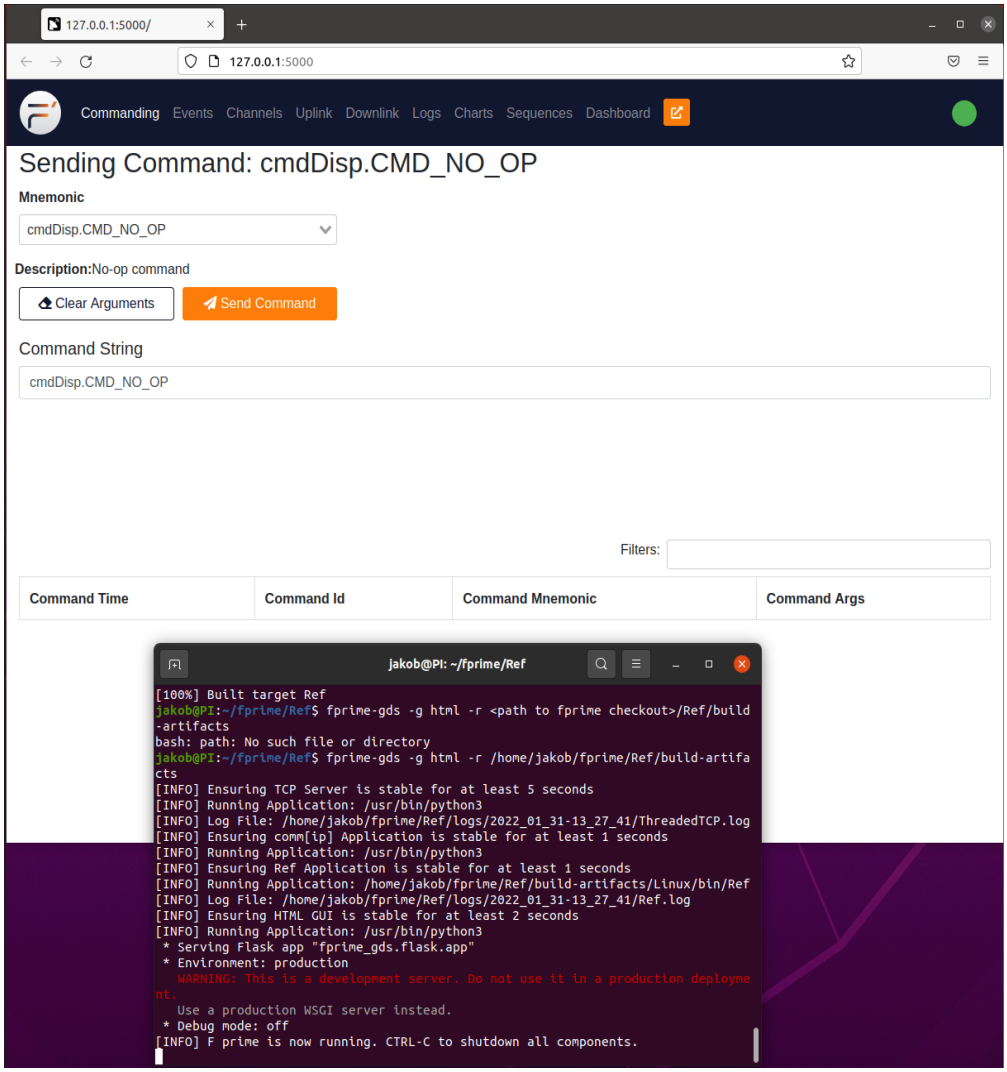


Figure 4.47: HTML GUI

From figure 4.47 one can see that the ground station has 9 different options. The commanding window will be used for giving commands. This window will be investigated further in the next test with a Raspberry Pi. It includes a Mnemonic command list as seen in figure 4.48. When making an own implementation of the framework, the different commands and signals one makes in a program will be executable here. This will also be visualized in the next test when commands for the Raspberry Pi will be added to the Mnemonic.

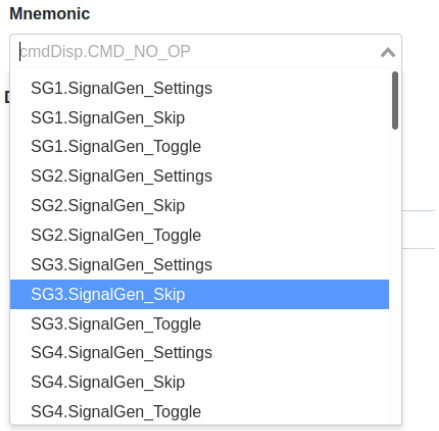


Figure 4.48: Mnemonic Command line

One can also see the other 8 different windows accessible. One can view the events in the event window, which visualizes executed events and signals and what kind of event it was. The Channels window shows the different active channels and their IDs. The Up and Downlink windows display the satellite communication from ground to satellite and vice versa. In the logs window, one can view the logs of different channels, which display how much memory and CPU power the different channels are using. The charts window gives one the opportunity to set up charts for different values, for example I/O values or sensor signals. The sequences window gives one the opportunity to make and execute command sequences. This by uploading a file to the GUI which contains the sequence. The dashboard window gives one the opportunity to upload an XML file to customize a dashboard. The windows are shown in the figures in Appendix C, with the exception of the Up and Downlink windows as they had no contents in this basic test.

4.10.3 Raspberry Pi with F'

The F' repository has an RPi folder, which is a test platform for Raspberry Pi. It contains the settings and topology to make an F' deployment on the Pi. Going through this tutorial will give practical experience using the framework, and give a more in-depth look at what is needed to deploy the framework on different devices. The flight controller in this case is a Raspberry Pi, so this is highly relevant if one wants to run F' on the flight controller. The operating system on the Pi is in this case the 64-bit version of the Raspbian Bullseye OS. How to flash the OS onto the Compute Module is shown in Appendix F.

Preparations

To run the test program, and also to use F' and Raspberry Pi together, SSH and RPi cross-compilation must be enabled and installed. To enable SSH on the Pi, execute raspi-config in the terminal on the Pi 4.49. Select Interface and enable SSH by selecting it and pressing enter twice. To install the RPi cross-compilation tools, use the Git repository from Raspberry Pi and clone this into the /opt folder on the host machine 4.3. It is also practical to set up a static IP address for the Pi, as this takes away the need for a monitor for the Pi to locate the IP address. This setup is shown in figure 4.50. Since the drones should work on wifi, this is the interface chosen for the IP address. Should one wish to use ethernet, replace the first line with $interface\ eth0$.

Listing 4.3: Raspberry Pi Tools

```
host@hostcomputer:/opt mkdir rpi
host@hostcomputer:/opt/rpi git clone https://github.com/raspberrypi/tools.git
```

Listing 4.4: SSH connection

```
pi@raspberrypi:~ sudo raspi-config
Press 3 "Interface Options"
Press I2 "SSH" and enter twice
```

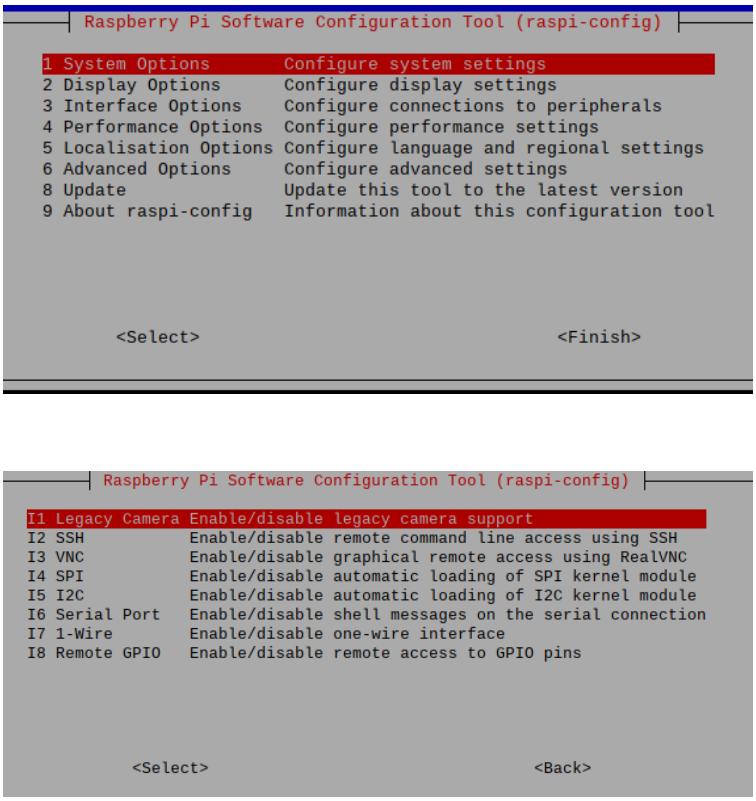


Figure 4.49: Raspberry configuration layout

```
  GNU nano 5.4
# A sample configuration for dhcpcd.
# See dhcpcd.conf(5) for details.
interface wlan0
static ip_address=128.39.202.11
static routers=128.39.202.1
static domain_name_servers=128.39.202.1 8.8.8.8
# Allow users of this group to interact with dhcpcd via the control socket.
#controlgroup wheel

# Inform the DHCP server of our hostname for DDNS.
hostname

# Use the hardware address of the interface for the Client ID.
clientid
# or
# Use the same DUID + IAID as set in DHCPv6 for DHCPv4 ClientID as per RFC4361.
# Some non-RFC compliant DHCP servers do not reply with this set.
# In this case, comment out duid and enable clientid above.
#duid

# Persist interface configuration when dhcpcd exits.
persistent
```

Figure 4.50: Static IP-address setup for RPi

Now for the program. This program is a basic blinking LED test, to test the connection between host and target and also that the target can output signal to externals. It is specified that this program communicates over port 50000 to the Pi. One must then make sure that this port is properly forwarded to the network, in both the VM and Windows, and that it is not blocked by the firewall. How to do this is explained in Appendix D.

Now that everything is set up, the test program can be deployed to the Pi. To make sure that the program works, a simple LED circuit is set up and connected to physical pins 39 (GND) and 40 (GPIO 21). This is shown in the simple schematic and physical setup in figure 4.51. To know which pins are which, one can either check the I/O board datasheet [101] on page 10 figure 5, or run the pinout command in the RPi terminal.

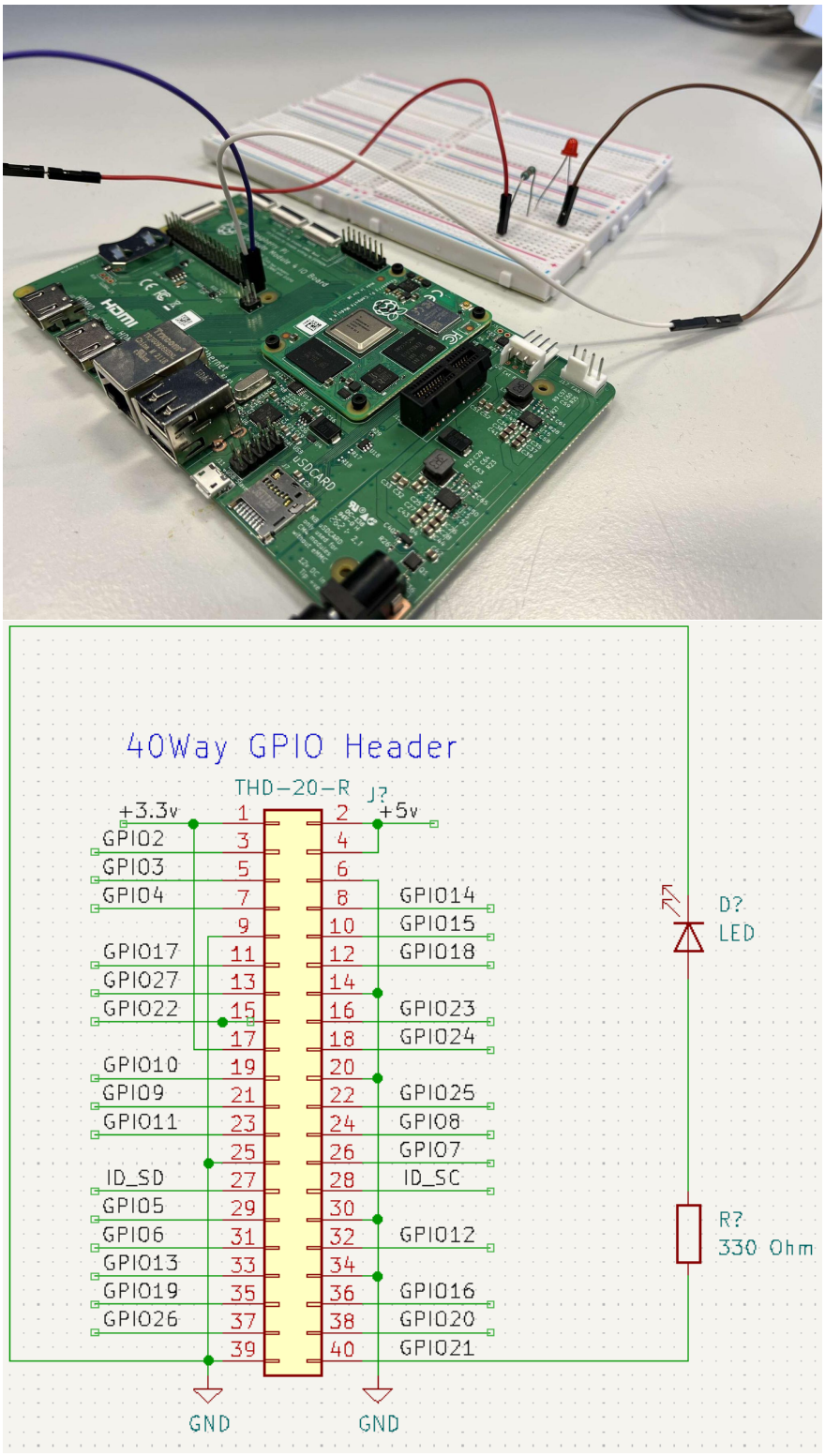


Figure 4.51: Physical test setup with schematic

Now cd into RPi in the fprime folder. First export the cross-compilation toolchain specified in the tutorial. This was cloned into the /opt/rpi folder on the Ubuntu, so execute the second command in listing 4.5 to invoke the correct compiler. Now generate the build directory for the example by executing line 3. This step uses CMake to copy all the necessary files from the F' source tree to the build directory. Once this step is done, the build process can start by executing line 4. This builds a binary file with the RPi example code, which will be transferred to the Pi. When the process is finished, this binary file is NOT placed where the tutorial says. Instead, it is placed deeper into the build tree, more specifically at /home/fprime/RPI/build-artifacts/raspberrypi/bin.

In the terminal, cd into the path shown above. From here, use the scp command to copy the binary file over SSH to the Pi. This is line 6. When this is completed, SSH into the Pi and locate the binary file. This should be in the home folder of the Pi if line 6 was executed.

Open another terminal and cd into fprime/RPI. From here execute line 7 in 4.5. This will start the ground station based on the examples topology.xml file. The topology introduces a few more commands specific to the RPi tutorial, which includes for example a command to turn the LED on/off and also for the LED to blink at a certain frequency. Execute line 2 in 4.6 and the binary file will boot up and the program will start. Now one can control the LED state and blinking frequency through the ground station.

Listing 4.5: Building and running the F' RPi application (HOST)

```
host@hostmachine:~ cd fprime/RPI
host@hostmachine:~/fprime/RPI export RPI_TOOLCHAIN_DIR= ...
    /opt/rpi/tools/arm-bcm2708/arm-rpi-4.9.3-linux-gnueabihf/
host@hostmachine:~/fprime/RPI fprime-util generate
host@hostmachine:~/fprime/RPI fprime util build
host@hostmachine:~/fprime/RPI cd
host@hostmachine:~/fprime/RPI/build-artifacts/raspberrypi/bin scp RPI ...
    pi@128.39.202.11:~
host@hostmachine:~/fprime/RPI fprime-gds -n --dictionary ...
    ./build-artifacts/raspberrypi/dict/RPITopologyAppDictionary.xml
```

Listing 4.6: Running the binary on the Raspberry Pi

```
host@hostmachine:~ ssh pi@128.39.202.11
pi@raspberrypi:~ sudo ./RPI -a <hostcomputer IP> -p 50000
```

It was also necessary to export a library path to the raspberry pi, due to the fact that the cross-compilation built a 32-bit executable, and the Raspbian OS on the pi is 64-bit. It was therefore necessary to envoke the 32-bit pthread library to run the program. This exportation is done as shown in the listing below. A 64-bit system is backward compatible with 32-bit, but not the other way around. The issue concerning the 32 vs 64-bit compilation is carried out in the discussion chapter, as a 64-bit interface would always give the best and fastest results to run a 64-bit program.

Listing 4.7: Exporting library path

```
pi@raspberrypi:~ export LD_LIBRARY_PATH=/usr/arm-linux-gnueabihf/lib
```

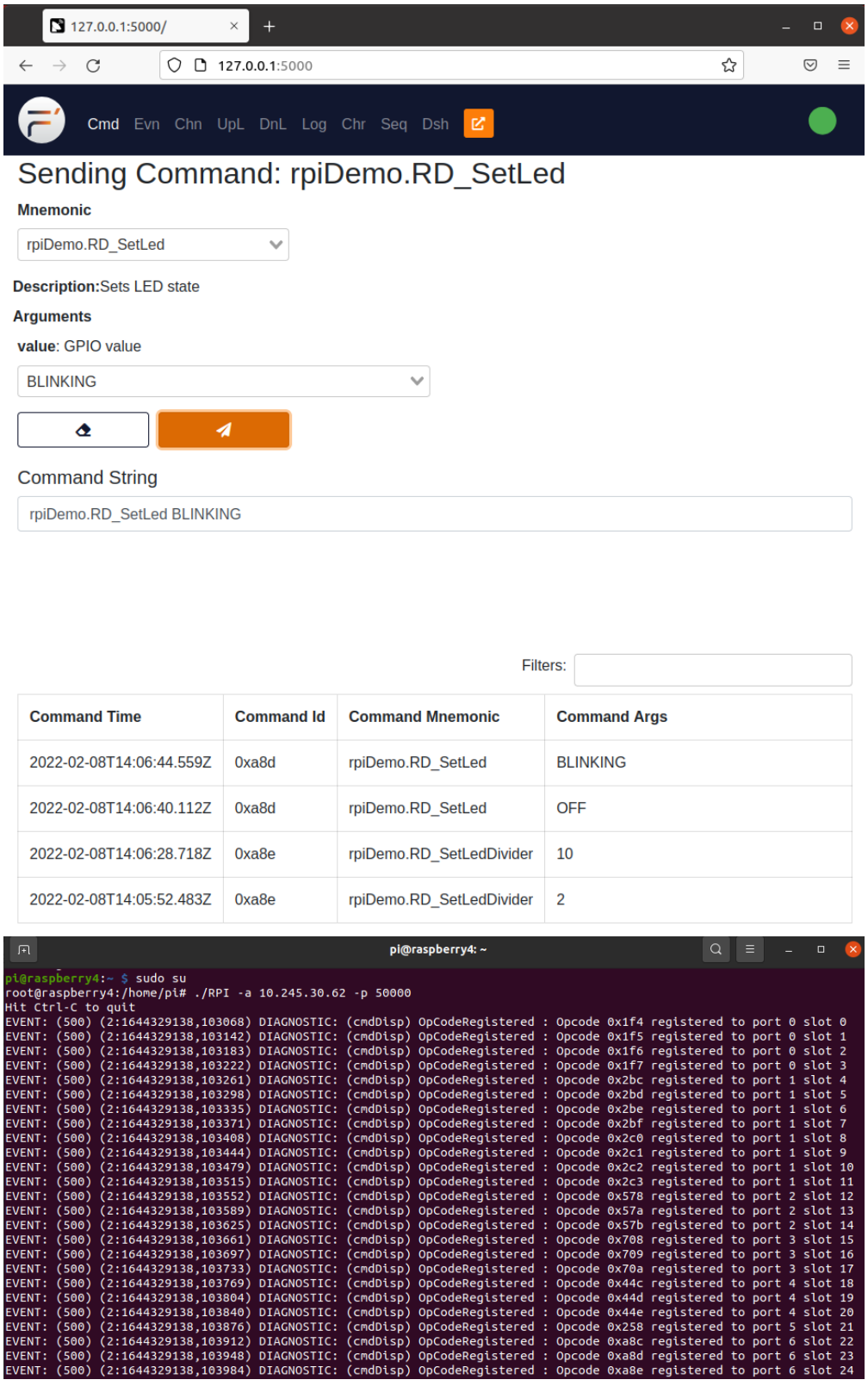


Figure 4.52: Launching of the ground system and RPi application

The LED should now be blinking, and by choosing the command rpiDemo.RD_SetLedDivider to for example 5 the LED should blink at 2 Hz and by changing it to 100 the LED blinks at 0.1 Hz as shown in the top figure in 4.52. This is because the rate is 10Hz divided by the argument put in. In the bottom figure in 4.52, the events are being logged in real-time. The RPIDemo can also be expanded, where more components and commands can be added, and is the suggested way of making a project for RPi in F' by NASA.

4.10.4 Component implementation in F'

The next step is to implement the sensors on the PCB as components in the F' interface. As mentioned in the previous section, NASA recommends that RPi projects should be implemented into the RPi demo folder, so this will be the initial condition for implementing the sensors. The F' staff provides a tutorial that implements a GPS into the framework. This tutorial is followed, with manipulations, to try to include the flight sensors. The tutorial can be found here [88].

The first step is to make a directory with the desired name inside the RPi folder of F'. We then make a directory called "Barometer", as we want to initially implement a barometer sensor. Originally this should be the BMP384, but as the PCB is not ready and there is no BMP384 soldered onto a breadboard, we utilize the MS5611 pressure sensor as a substitute. This communicates over I2C as opposed to the BMP-sensor that uses SPI. This needs to be changed if the implementation process is successful, but for test purposes, the MS5611 will suffice. This is connected to the RPi I/O board on the default I2C pins. These are GPIO pins 2 and 3. (bilde kanskje).

```
host@hostmachine:~ cd fprime/RPI
host@hostmachine:~/fprime/RPI mkdir Barometer
host@hostmachine:~/fprime/RPI cd Barometer
```

Now the first step is to make a .xml file with the desired name. This file will contain all the drivers and ports required to build the component. This includes different drivers from the framework, such as serial and in our case I2C drives. This file is shown in the Appendix figure E.1. The drivers are located in the Drv folder in the F' framework. Inside there are different modules for the drivers. The port driver for the I2C is shown below 4.53. It has two parameters. One is the I2C address of the sensor, and the other is a buffer that contains the read/write information. In the code, these have to be imported and implemented and are done so under the "import_port_type" section and the "Telemetry ports" section of E.1.

```
24 module Drv {
25
26   port I2c(
27             addr: U32 @< I2C slave device address
28             ref serBuffer: Fw.Buffer @< Buffer with data to read/write to/from
29           ) -> Drv.I2cStatus
30
31 }
```

Figure 4.53: I2C Port driver

As shown in the last three lines under the "import_port_type" in E.1, the three needed dictionaries are implemented. These are the command, telemetry, and event dictionaries. The command and event dictionaries are represented in the GUI of F', as shown in figure 4.47. This is where one can implement commands to for example read the registers of the sensor or as in the RPi LED example, send a blinking command to an LED.

These dictionaries have to be created for the pressure sensor, as the Ai.xml file was. The command file is manipulated to hold one command, which is to report the status of the barometer. This is done to make sure that the barometer is online and streaming data over I2C. This is shown in E.2. The telemetry file is manipulated to give three channels of information to the GUI. This is the pressure and temperature in both Celsius and Fahrenheit given by the sensor. This is shown in E.3. The last file, the event file, contains two events. It reports if the barometer connection is online or offline. This is shown in E.4.

F' uses, as mentioned earlier, CMake to build projects. This means that in order to build the barometer application, a CMakeLists file must be created and list the different source files to be built. In the CMake file for the barometer folder, only one line is needed. This is shown in 4.8. It includes the Ai.xml file made in the first step. Then another CMakeList should be made, but since we are in the RPI folder, it already has one. Therefore we implement our subdirectory, being the barometer folder, into the RPI CMakeList. The complete list is shown in E.5, and the line which should be added is also shown in figure 4.9. This CMakeList is the basic list provided by the F' framework, and implements the core framework components such as the framework path and the F' code.

Listing 4.8: CMakeLists.txt for the barometer folder

```
####
# GPS Tutorial: GpsApp/Gps/CMakeLists.txt ...
    (BarometerApp/Barometer/CMakeLists.txt)
#
# SOURCE_FILES: combined list of source and autocoding files
# MOD_DEPS: (optional) module dependencies
#
# This file will set up the build for the barometer component. This is ...
    done by defining the SOURCE_FILES variable and then
#registering the component as an F prime module. This allows autocoding ...
    and more!
####
set(SOURCE_FILES
        "${CMAKE_CURRENT_LIST_DIR}/BarometerComponentAi.xml"
)
register_fprime_module()
```

Listing 4.9: CMakeList of the RPI application

```
#Add component subdirectories
add_fprime_subdirectory("${CMAKE_CURRENT_LIST_DIR}/RpiDemo/")

# Add topology subdirectory
add_fprime_subdirectory("${CMAKE_CURRENT_LIST_DIR}/Top/")

#Add barometer subdirectory
add_fprime_subdirectory("${CMAKE_CURRENT_LIST_DIR}/Barometer/")
```

To now build the project with the barometer implemented, run the fprime-util generate command. This should first be done inside the barometer directory, as it builds the header and cpp file of the Ai.xml. This is needed to later run the same generate command in the RPi directory. When the header and cpp file for the Ai.xml is created, we then run the fprime-util impl command to make implementation files for the barometer. Inside the generated header file, the framework generates some basic functions and classes based on the .xml files created earlier. The user can now implement their own handler functions to for example read the barometer values. These are then used in the generated cpp file under the TODO lines as shown in 4.55.

As shown in figure 4.54, there are two functions. The one on the top is from the tutorial example, which reads through serial communication, and the bottom one is the I2C read function we have implemented. It takes in the port number and uses the buffer to store data for the read operation. The I2C status driver is called to provide information on the sensor status. This is based on the code in figure 4.53, where the port number will be the I2C address, the buffer is the same, and it reports the outputs to Drv.I2cStatus.

```
PRIVATE:

  // ----------------------------------------------------------------------
  // Handler implementations for user-defined typed input ports
  // ----------------------------------------------------------------------

  //! Handler implementation for serialRecv
  //!
  void serialRecv_handler(
      const NATIVE_INT_TYPE portNum, /*!< The port number*/
      Fw::Buffer &serBuffer, /*!<
  Buffer containing data
  */
      Drv::SerialReadStatus &status /*!<
  Status of read
  */

  );

  void I2cRead(
       const NATIVE_INT_TYPE portNum,
          FW::Buffer &serBuffer,
          Drv::I2cStatus &status
  );
```

Figure 4.54: Function declarations in the header file

The function made above is now to be implemented in the cpp file. There are two different addresses of the sensor. If the chip-select pin is grounder, the address is 0x77 and if the VCC pin is connected, the address is 0x76 as stated in the datasheet [36][105]. This sensor is connected to the VCC pin, so the address used is 0x76. This is written into the function, and the buffer and status remain the same.

```
// ----------------------------------------------------------------------
// Handler implementations for user-defined typed input ports
// ----------------------------------------------------------------------

void barometer ::
  serialRecv_handler(
      const NATIVE_INT_TYPE portNum,
      Fw::Buffer &serBuffer,
      Drv::SerialReadStatus &status
  )

  // TODO
void Barometer::I2cRead(
        const NATIVE_INT_TYPE 0x76
        FW::Buffer &serBuffer
        Drv::I2cStatus &status

        )
```

Figure 4.55: Function implementation in the cpp file

Now that the implementation is finished, the CMakeLists file in the barometer directory has to be updated to include the cpp implementation file. We write this under the existing line in the source files as shown in 4.10. After this, the configurations are finished, and we run the fprime-util build command to build the topology for the project.

Listing 4.10: Updated CMakeList in the barometer directory

```
set(SOURCE_FILES
        "${CMAKE_CURRENT_LIST_DIR}/BarometerComponentAi.xml"
        "${CMAKE_CURRENT_LIST_DIR}/barometerComponentImpl.cpp"
)
register_fprime_module()
```

The rest of the tutorial is based on using the Ref application to build the rest of the project. As we are in the RPi directory and the LED blink project already has been built, this is not needed. The next step is to initialize the GUI and test if the sensor is working and can be operated through the GUI! The binary executable is located in the bin folder of the build-artifacts directory. Since we are using the RPi LED application, the topology file is located in the same place, so to launch the GUI the command in 4.11 is used. This is where the project ran into some problems. This is further discussed here 6.6.

Listing 4.11: Launch of the GUI

```
host@hostmachine:~/fprime/RPI$ fprime-gds -n --dictionary ...
    ./build-artifacts/raspberrypi/dict/RPITopologyAppDictionary.xml
```

4.11 Programming the flight controller

Originally, the plan was to use Simulink as the intended platform for most of the code work, as it is a familiar program and also very user friendly because it utilizes blocks instead of direct code writing. It was early discovered that Simulink would not work on the Raspbian OS, as when trying to set it up, Simulink was not able to download the correct libraries to the Pi. It was tried to use the predesign Raspbian OS from Matlab with Simulink already set up, but this Raspbian kernel version was 5.4 which is much older than the one we use. There was also an issue of how the Simulink programs were to be included in the F' framework, and because of all these complications, it was decided to not use Simulink at all and rather use C/C++ to code the programs.

As the first iterations of the PCB were produced, the procedure of setting up the sensors started. Both the IMU and Barometer are provided by Bosch Sensortec, so to set up the IMU and Barometer to receive readings, the initial step is to look in the Bosch-API for the sensors. These are located in two repositories on the Bosch GitHub page. For the IMU [27] and for the barometer [28].

4.11.1 Setting up the RPi SPI-channels

Setting up the different SPI-channels on the Raspberry Pi requires changes in the config.txt file in the boot folder. There are in total 6 SPI-channels on the Compute Module 4 as shown in figure 4.15. We are using SPI-channel 1 for both the IMU and barometer, so this has to be implemented in the config file to activate the channel. The IMU uses two chip selects and the barometer uses one. There the following lines have to be added 4.12. Upon a reboot, these settings have been set.

Listing 4.12: Setting the SPI-channels

```
dtoverlay=spi1-3cs, cs0_pin=18, cs1_pin=17, cs2_pin=16
```

4.11.2 Reading the IMU

The Bosch-API provides a setup procedure to initialize communication with the BMI085 sensor. This is described in the DataSynd.md file in the repository, and initially, this was the procedure used. It was quickly realized that the example programs in the repository were indeed directed towards data extraction from the sensor, but to run these required a shuttle board capable of communication with COINES. COINES is a software package developed by Bosch for their sensors, and this introduced a need for a shuttle board that was not available at the time of the project. In addition, it is not listed that COINES supports the Raspberry Pi platform, and therefore this would be a waste of time. It was initially tried to implement the tutorial from Bosch, but it was quickly realized that it was not going to work. This was due to all the dependencies of the COINES software in the repository, and maneuvering around them was quite difficult. It was therefore decided to use SPI-communication libraries for the Raspberry Pi to attempt to read the sensor values.

There are several SPI libraries compatible with the Raspberry Pi platform. The most known ones are "Wiring Pi" [53], "pigpio" [60], and also the default SPI-library on the Pi "spidev". The spidev library can be used together with the IOCTL system call to control input and output operations on the Pi. The spidev approach is a device-specific approach compared to the other libraries [97], [126], [125]. All of these approaches are tested, as it was shown that communicating with the sensor was not an easy task in this particular setting.

Wiring Pi

The Wiring Pi approach has already been used in the bachelor thesis of 2019, [72], although it used the BMI088 sensor, whereas this project will have to use the BMI085. In 2019 the I2C-protocol was used to communicate with the sensor, whereas this project will use the SPI-protocol. Fortunately, the Wiring Pi library includes both I2C and SPI libraries, so the initial approach was to make a program based on the previous version from 2019 and substitute the I2C-functions with SPI-functions and also adjust the registers to the projects needs.

The BMI085 sensor always starts in I2C-mode, and to switch this to SPI, a rising edge must be detected on the accelerometer CS-pin as mentioned in chapter 3 in the datasheet [23]. The accelerometer CS-pin is connected to GPIO 17, as mentioned in 4.4.5, and we use the digitalWrite function from the library to set the CS-pin high. The GPIO pin selections are defined at the top of the code to keep a tidy code layout. When the rising edge has been sent on the accelerometer CS-pin, the IMU should switch to SPI, and the gyroscope does not need this as it automatically selects its protocol based on the PS-pin. Since this PS-pin is grounded, the gyroscope starts in SPI-mode.

To perform read/write operations with the sensor, the CS-pins must be set to low to indicate that a read/write operation is happening. From there the desired registers one wishes to read from or write to must be sent to the sensor, where the sensor will replace the written values with the data from the written register. In chapter 6.1.2 of the datasheet [23], there is a description of reading the accelerometer registers. This says that when a data request is sent to the sensor over SPI, the value will not come right away, but we will receive a dummy byte first and the next bytes will be the requested data. To write to a register the first bit in the write operation should be either 0 or 1, where 0 indicates a write and 1 indicates a read. The next 7 bits should be the register address we wish to read from or write to, and bit 8 to 15 is either the data we wish to write to the address or unpredictable data when reading. Therefore one should read bit 16-23 to get the desired data when reading.

Register address							1/0	Data SDI for writing							
x	x	x	x	x	x	x	R/W	Dummy byte	x	x	x	x	x	x	x

Figure 4.56: Writing to the sensor through SPI

Register address							1/0	Unpredictable data							
x	x	x	x	x	x	x	R/W	Dummy byte	x	x	x	x	x	x	x

Readable data							
x	x	x	x	x	x	x	x

Figure 4.57: Reading sensor output through SPI

This means that when we are writing a register, the first bit should be 0 or 1 followed by the address. To do this, the first bit is set to the desired value and then the address, which is 8 bits, must be shifted one bit to the left. This is shown in the Appendix code G.1, but for an example, we can take the LSB address of the accelerometer's x-value which is at address 0x12, shift it one bit to the left and add the read command 4.13.

Listing 4.13: Read operation with Wiring Pi

```
#define BMI08X_REG_ACCEL_X_LSB 0x12

float x;
unsigned char buf[10];
buf[0] = (BMI08X_REG_ACCEL_X_LSB << 1U) | 1U;
buf[1] = 0x55; // This is the dummy byte
buf[2] = 0x00; // This is the byte that will contain the sensor readings

x = (int16_t)wiringPiSPIDataRW(SPI_CHANNEL, buf, 3);
```

As mentioned, the CS-pin should be set low to initiate a transfer. The Wiring Pi library takes care of this inside the DataRW function, so there is no need to specifically set the CS-pin low before each transaction. The DataRW only takes 3 parameters, which are the SPI channel, buffer with data, and length of the buffer. In the first iteration of the code, the bachelor thesis from 2019 was used as mentioned, as a reference. Had implemented its own function for transferring the desired configurations over I2C. This function was used but tweaked to fit SPI. There is though an error in this, which was not noticed until later on. In the function, the SPI only writes the configuration of the register, and not a buffer containing the register and the configuration we wanted to do in it. This is shown in G.1 in the function regConfig. We then had the issue that the IMU gave zero values, which could be because the sensor had not been properly switched to SPI and in addition, the accelerometer has to be woken from sleep mode using the PWR_CTRL register. The code was thoroughly tested, and after a long time with only zero values, it was decided to try and use a different library.

Pigpio

The second library that was tested was the pigpio library [60]. The library provides both regular read and write operations over SPI, and also bit-banging. Bit-banging often has a noticeable effect on the real-time system's ability to meet deadlines, as it consumes a lot of CPU power [73]. Although bit-banging is not ideal in a real-time environment, this was initially tested to check if the library was able to retrieve values, as up till this point there was no success reading the sensor data. The code is shown in Appendix G.2.

The bit-bang portion of the library has three functions. One to open the SPI-device, one to read or write to the SPI-device, and one for closing the device. These are named "bbSPIOpen", "bbSPIXfer", and "bbSPIClose" respectively. The bbSPIOpen function demands 6 input parameters. The first four are the pins the SPI-device is connected to, ergo CS, MOSI, MISO, and SCLK. The fifth parameter is the baud rate, which is defined to 2 MHz as the datasheet suggests. The last parameter is called spiFlags 4.58. This is an unsigned 22-bit input, which for example defines the SPI-mode we want to use. There are four modes ranging from 0 to 3, and we want to use mode 0 which is the default. To open both the accelerometer and the gyroscope, two instances of the SPIOpen function has to be called, and defining the CS pins for the accelerometer and gyroscope respectfully. In the code, this is only done for the accelerometer for test purposes.

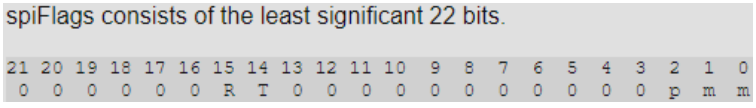


Figure 4.58: SPI flags from pigpio library [60]

The second function is used similar to the way Wiring Pi uses the DataRW function. In pigpio the CS pin has to be specified in the R/W function as well as a buffer for writing and another buffer for receiving. The last parameter is the length of the write buffer. To do a write or a read, one would still have to shit the bits of the address one time to the left and use a zero or a one as the first bit. This would look like the representation in 4.14 below. As in this instance the code only writes to the SPI-device, there is no need to add 3 bytes to the buffer as there is no read operation involved.

Listing 4.14: A write operation using pigpio

```
#define BMI08X_REG_ACCEL_PWR_CTRL 0x7D
#define REG_ACCEL_PWR_CTRL 0x04 // Powering on the sensor
#define CS1 17

unsigned char inBuf[2];

char t2[] = {(BMI08X_REG_ACCEL_PWR_CTRL << 1U | 0), REG_ACCEL_PWR_CTRL};
tx = bbSPIXfer(CS1, t2, (char*)inBuf, 2);
```

Also using this library, the received values were either zero or a static number. This led us to try to code a program specifically for the CM4 module, which used the default SPI library on the Raspberry Pi. This library is called spidev [97], where the IOCTL system call can be used to make sure that the CS-pins are set high and low, and also to ensure that the messages over SPI go directly to the designated pins the IMU is routed to, on the PCB [126].

Spidev and IOCTL

These are the last libraries that were used to try and retrieve the IMU data. It is, as mentioned, a device-specific approach. This is because it uses a system call to communicate with the Linux kernel to execute a service. As this article describes, "*System calls are the primary method through which user-space processes call kernel services*"[16]. Together, all these systems call to form an application user interface (API) directly linked to the operating system. In addition to this, it is worth mentioning that the previous system call to control a GPIO pin has been changed from the sysfs interface to a new GPIO interface. To include this in the code, the header and c file of the gpiolib are retrieved from the Linux drivers repository on their GitHub page [124] and put into the folder containing the code. The code inspiration comes from a Youtube tutorial provided by EmbeddedCraft [43].

The start of the code is the same as before in addition to the spidev, IOCTL, and gpiolib header files. The code is shown in Appendix G.3. To set the CS-pins for the accelerometer and gyroscope, a function has to be made. This is named "set_gpio_pin" in the code with two input parameters. These are the GPIO pin and the desired direction. First, the chip select GPIO pin is exported and then set to be output.

Secondly, a write function for the GPIO pins have to be made to be called in the code to set the CS-pins high or low depending on if a transaction is occurring or not. This is named "write_to_gpio" and also takes in two parameters. One is the GPIO pin number and the last one is its value.

The next step is to export the correct SPI-device from spidev. Since the IMU is connected to SPI-channel 1 and the accelerometer is using CS1, the device we want to open is "spidev1.1". For the gyroscope, this would be spidev1.0 as it uses the same SPI-channel and CS0 which is GPIO 18. To initiate the device, we start with defining the SPI-mode we want to use, which is 0, and the baud rate which is 2MHz. From there we open the SPI-device with reade and write permissions. Now we use the IOCTL system call to set the SPI-mode and baud rate. The spidev header file

contains a struct in which one can define the SPI-transfer parameters. It contains several parameters to be set, but we only use the ones we need. The struct is named "xfer", and to view the parameters set, take a look at the code in G.3. An example of how to open the SPI-device can be viewed in 4.15.

Listing 4.15: Opening SPI-device with R/W permissions

```
int val = 0;
int iSPIMode = SPI_MODE_0;
int spi_freq = 2000000; // 2MHz

int f_spi = open("/dev/spidev1.1", O_RDWR);

val = ioctl(f_spi, SPI_IOC_WR_MODE, &iSPIMode); // Setting the mode
val = ioctl(f_spi, SPI_IOC_WR_MAX_SPEED_HZ, &spi_freq); // Setting the ...
   baudrate
```

The last function to be implemented is the read/write function. This function is named "spi_write" in the code. Inside the code, its defined to set the CS-pin low before we initiate the transfer. From there, the transmit and receive buffers are set to the appropriate settings and finally the IOCTL system call is used to transfer the buffer. The variable which contains the message is called "SPI_IOC_MESSAGE(N)". To do the transfer, one must call IOCTL and provide three input parameters. The first is the SPI-device, the second one is the variable containing the message and the last is a reference to the spi_ioc_transfer struct. After this, the CS-pin is set to high to indicate that the transfer is complete and the device is ready for another transfer. An example of how to do a SPI-transfer using IOCTL is shown in 4.16;

Listing 4.16: Initialising a read/write transfer using IOCTL

```
int val = 0;

val = ioctl(f_spi, SPI_IOC_MESSAGE(1), &xfer);
```

As mentioned when using the previous libraries, the address to write to or read from must still be shifted to the left and the first bit must indicate a read or a write. This is shown in the code in G.3. At the initial stages of testing this code, we still received zero values. This led us to first use the I/O board with an LED and then an oscilloscope to show if the CS-pin was set high and low when demanded to.

4.11.3 Debugging the IMU

Using the I/O board and an LED

The first step in debugging the IMU connections was to put the CM4 on the I/O board and use the gpiolib to set the CS-pins high and low. Then an LED is connected to the CS-pin, and upon receiving high and low output from the CS-pin, the LED should blink. To do this, we write a program that uses the gpiolib to output the signal, as this is the way it is done in the original code. A simple while loop can be made, which turns the output signal high and low with a 1-second interval. This is shown in the code below 4.17.

Listing 4.17: Test code for the CS-pin with LED

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include "gpiolib.h"

int main()
{
    int pin_number = 18;
    gpio_export(pin_number);
    gpio_direction(pin_number, 1);

    while(pin_number == 18){
    gpio_write(gpio_pin, 1);
    sleep(1);
    gpio_write(gpio_pin, 0);
    sleep(1);
    }
return 0;
}
```

This test resulted in a blinking LED when running the program, so this indicates that the library is capable of outputting a signal to the CS-pins. To test this even further, an oscilloscope was used.

Using the oscilloscope

To be able to troubleshoot the circuit where the IMU was placed (Figure 4.16) it was necessary to carefully remove some of the paint that covered the tracks leading out of the IMU and to the HIROSE connector. It was important to not use a tool that was neither too big or too powerful, because of the risk of cutting the thin cobber track. After the paint was removed, it was still not a big enough clearance to get the oscilloscope pin to properly hit the cobber track. The solution was to use a small amount of solder paste and use a heat gun to create a small solder pad over the cobber track. This allowed the oscilloscope's test pin to get a connection with the signal from the IMU CS-pin. Then it was possible to run the test again and be able to see when it goes low or high when running the test explained in section 4.11.3. In figure 4.59 it is pictured the testing pad used to be able to test the IMU CS-pin inside the small yellow circle.

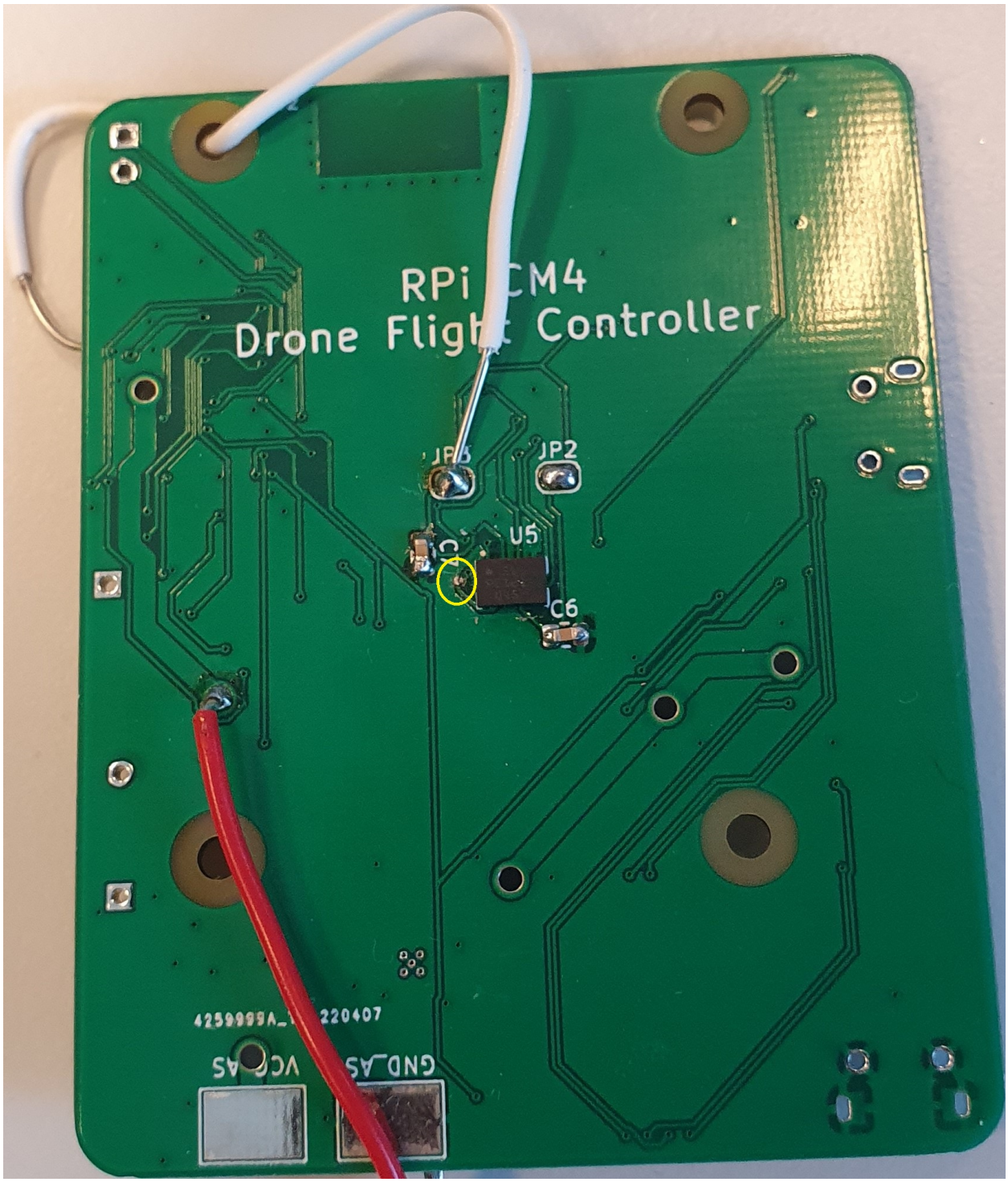


Figure 4.59: Backside of the PCB used to test the IMU

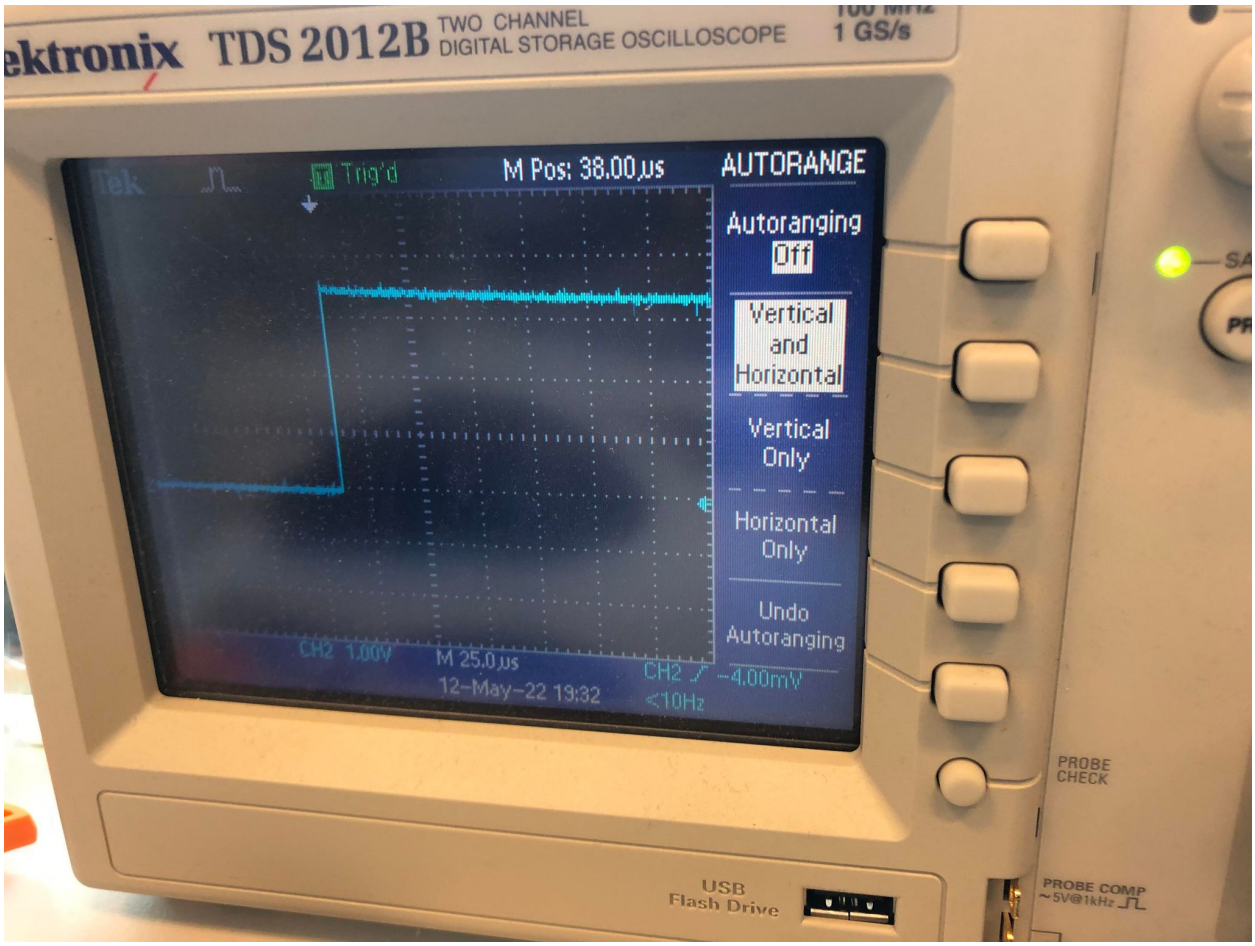


Figure 4.60: Measurement displayed on the oscilloscope

As one can see in the figure above, the oscilloscope shows the CS-pin output. It clearly sets the CS-pin high when we want it to, so this indicates that the functions to set the pin high and low in the program are working. This test was done both with wiring pi and the gpiolib, and both tests resulted in the CS-pin reacting to the input.

4.11.4 Testing the BMI085 shuttle board

Towards the end of the project, a BMI085 shuttle board was ordered as it became available on Mouser. The shuttle board contains the IMU with designated pinouts that can be connected with jumper wires. The IMU could now be tested with reassurance that the pin paths were properly routed to the pins, as this is very hard to debug on the PCB. Even with these pinouts, there was a problem. The spacing between the pins was too small, which resulted in the jumper wires not fitting. The shuttle board came with a rather nontraditional pin-spacing, and the spacing was so small the only solution was to solder the connections. The soldering was first done according to the SPI-specifications on the board, which required 9 pins as shown in the datasheet [113]. This process was extremely difficult, even with the precision soldering iron. The pin spacing can be viewed in figure 4.61 and the soldering process in figure 4.62.

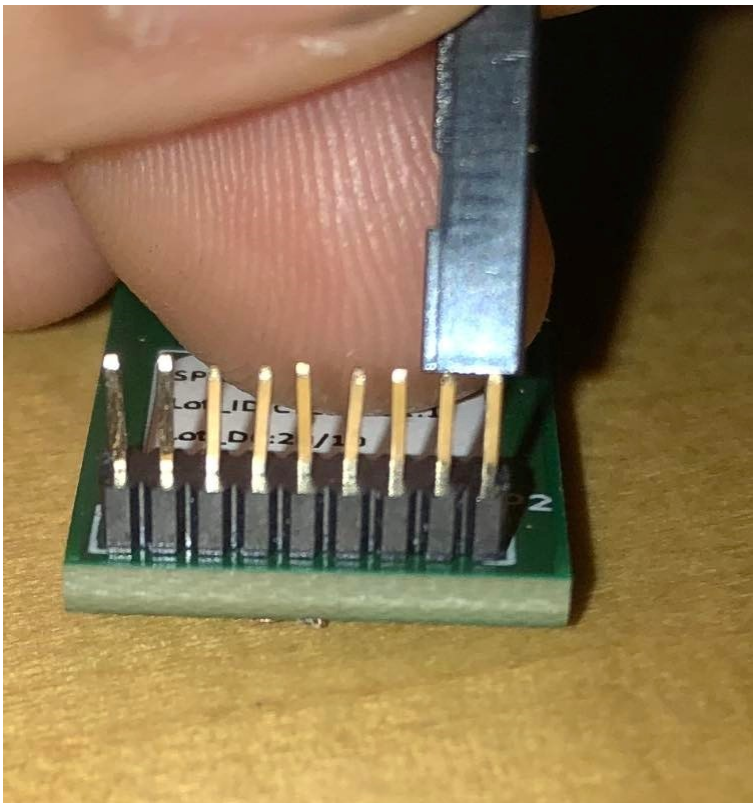


Figure 4.61: Picture of the pin spacing of BMI085 shuttle board

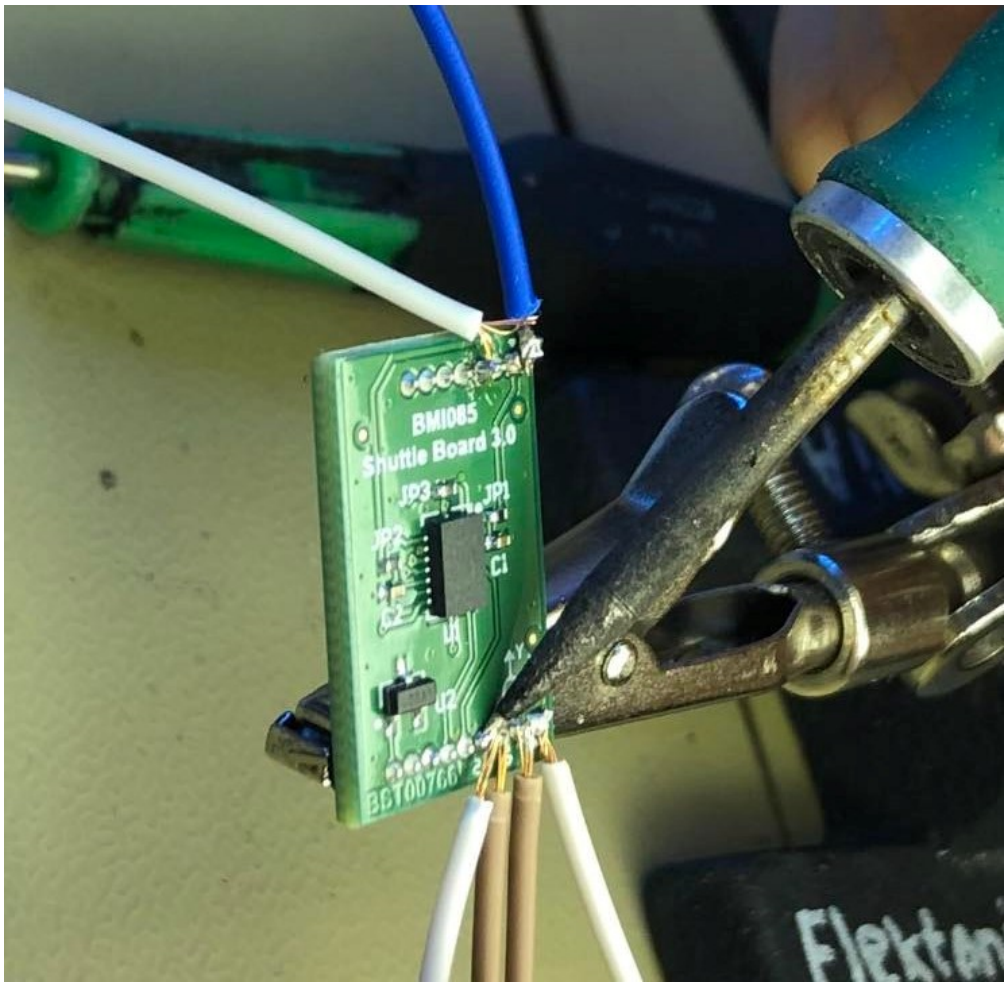


Figure 4.62: Soldering of the shuttle board

As one can see, the pin spacing is very small. The finished soldering product was tested with the code shown in G.4 and G.5, with the sensor connected to SPI-channel 0, but still no success. This led us to take off the solder, as it was believed that there were overlaps. We then tried to take off the caps of the jumper wires and use a crimping plier to make the cables fit. The connections were then coated with electrical tape to prevent shorting. This worked to a certain degree, but the spacing of the pins was still too small, as putting on more than one wire after another resulted in bending the pins. We then tried to access the IMU by switching the interface to I2C. This only required 4 pins, and these pins were not next to each other, so the risk of bending the pins was low.

When accessing the IMU through I2C, the IMU responded with the values requested. However, when moving the IMU around the connection was lost, when it was moving too fast. This was a result of the jumper wires being too big for the pins. We then soldered these four pins instead, and this gave us a working IMU that gave us the readings we wanted. This lead us to believe that when working with the SPI-interface, the pins were not connected properly, and thus we got no readings. For further testing, smaller jumper wires would have been ordered, to ensure that the pins were properly connected. There were no such wires available at the time of testing. The setup and result can be viewed in figure 4.63, and the code in G.6.

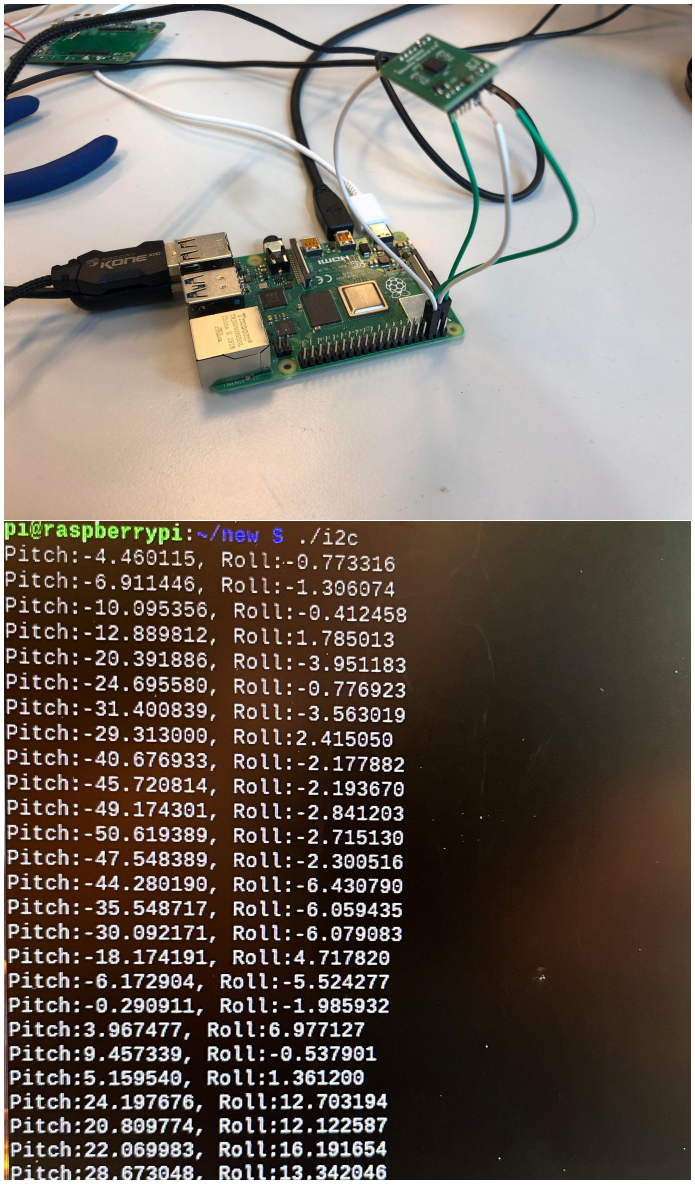


Figure 4.63: Shuttle board connected to RPI over I2C

4.12 Debugging of bricked Raspberry Pi CM4

Towards the end of the testing phase, the RPI CM4 suddenly would not boot anymore. When fitted to either the IO board or the V3 of the PCB it would only make a faint buzzing/clicking sound from the CM4 itself, and would not start. While being certain that it had not received too high voltage we checked both the IO board and the V3 to see if any other components were fried or broken. This was not the case. V2 of the PCB did previously have problems with a malfunctioning step-down voltage regulator which did not work (Figure 4.29). The RPI was never mounted while the voltage regulator was not working as the PCBs were always controlled that they were supplying the correct voltage before mounting the RPI. As too much voltage from the power supply was ruled out, both the IO board and the PCB V3 were checked for faulty power supply components to see if one of them could have fried the RPI, but all components were intact and the boards were supplying the correct voltage to the RPI.

After concluding that neither the PCB V3 nor the IO board had problems regarding power supply, a search on the internet after similar cases were conducted. It seemed it was not a widely occurring problem but it had happened to a few others while using the same IO board and another third-party breakout board, although their breakout boards were from trusted manufacturers such as Adafruit. This only reinforced the thought that it might not have been a triggering fault in our boards at all, and it might only be a problem in the RPI CM4 itself. After reading the forum thread [39] on possible debug methods to check whether the RPI was fried or only had boot issues, the RPI was mounted on the IO board and connected to power. While it was connected to power, and making the clicking/buzzing sound, the voltage on the VCC 5v and 3.3v output on the GPIO pins were measured using a multimeter. The 5v pin gives 3.9v out, only varying slightly between 3.85v and 3.98v. The 3.3v pin gives out between 1.38v and 1.41v. Some users hint at broken ferrite inductors on the board, and others that some component regarding the power supply had been broken, causing too much voltage to be delivered to the board. The last one has been ruled out. There is though a possibility that the pins on the HIROSE might have been too close to each other, making a short between two pins that might not often be in use, and therefore it happened a while into testing. This suspicion is due to the extremely small pins on the HIROSE, making debugging quite hard. The plastic on the connector also covers a lot of the visible pads, so it is sometimes not visible, even with a microscope. Based on the measurements of the voltage on the GPIO pins when powering the RPI it is quite certain that the RPI is bricked. Since the loss in voltage on GPIO pins indicates that the eePROM memory does not get the right amount of power or voltage, it is likely that this is the reason it will not boot, and is therefore broken [39].

4.13 Raspberry Pi and ATmega communication

Since the CM4, unfortunately, became bricked, there was no way of continuing the process of retrieving the IMU data. Therefore the next task at hand was to make the Raspberry Pi and ATmega microcontroller communicate over SPI. We used the Raspberry Pi 4 Model B to do these tests, as the CM4 is only a stripped-down version of the Model B. They share the same SPI-interface and same GPIO layout.

The microcontroller used was an Arduino Nano with the ATmega328P chip [6]. The code developed for the Arduino is also compatible with the ATmega32U4, as the only change needed is to select the appropriate bootloader. The Arduino Nano uses the 328P bootloader, while the ATmega32U4 uses the Lenardo bootloader [5].

The first test using the SPI-interface from RPI to Arduino was done before the CM4 got bricked. The I/O board was used to connect the Arduino to the SPI-pins, and the test was done based on the tutorial provided by "*roboticsbackend*" [15]. The test uses the Wiring Pi library on the master side (RPI) and the Arduino IDE on the slave side (Nano). The code can be found in Appendix G.7. The code is extracted directly from the source to make sure that it works according to the tutorial. The Arduino is connected to SPI-channel 0, shown in the left figure of 4.65. This is because it is on that channel the ATmega32U4 is connected to the PCB.

The program works as follows. The master side sets up the correct SPI-channel and speed of the process. It creates a buffer with two variables to be written to the slave. The first is the number 23 and the second is zero. The goal is to send the buffer to the slave, and have the slave increment the first number in the buffer by 10, and then return the incremented value to the master. The result is shown in figure 4.64. The first test sent the number 23 to the slave, and the Pi received 33. Which is the correct value. Next the number 33 was sent, where the received number was 43, which is also the correct value.

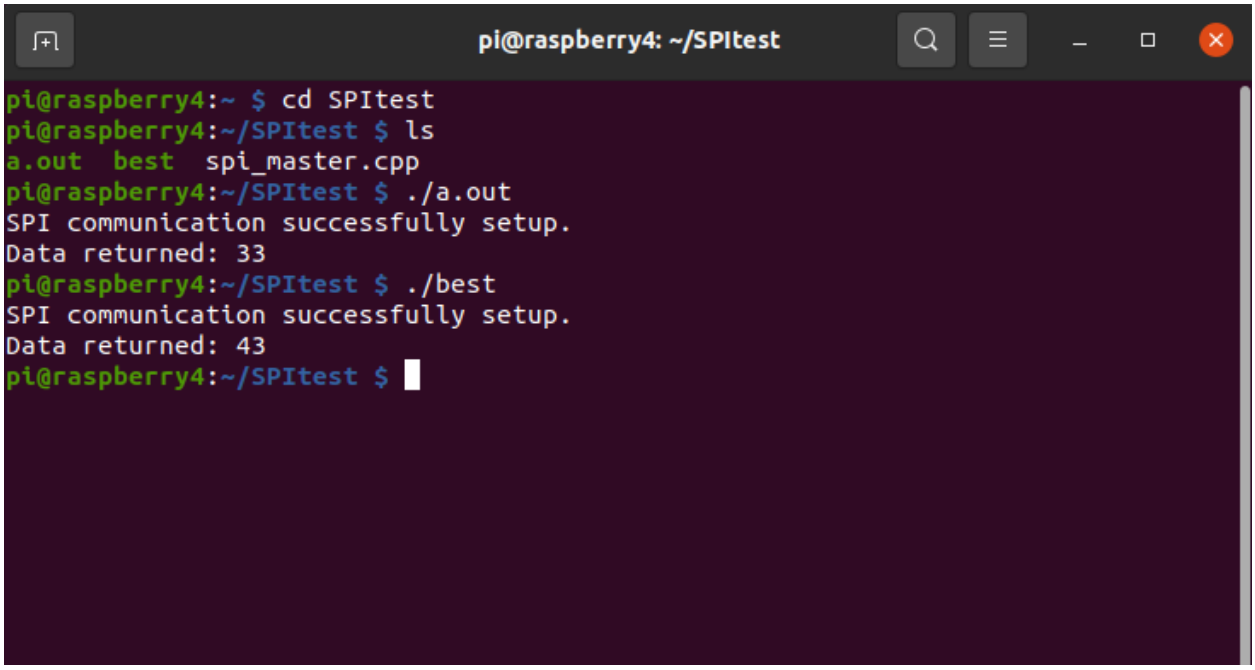


Figure 4.64: Result of the SPI test

The next step is to try to send some IMU values over SPI. We use the Wiring Pi library again, as this worked in the previous test. We also use an MPU9250 IMU, which comes on a breakout board. The pins can then be directly connected to the GPIO pins on the Pi. This IMU operates over I2C, but the values will be sent over SPI to the Arduino. A note here is that we are now using the Raspberry Pi 4 Model B, as the previous test was done before the CM4 was bricked. The setup can be viewed in figure 4.65.

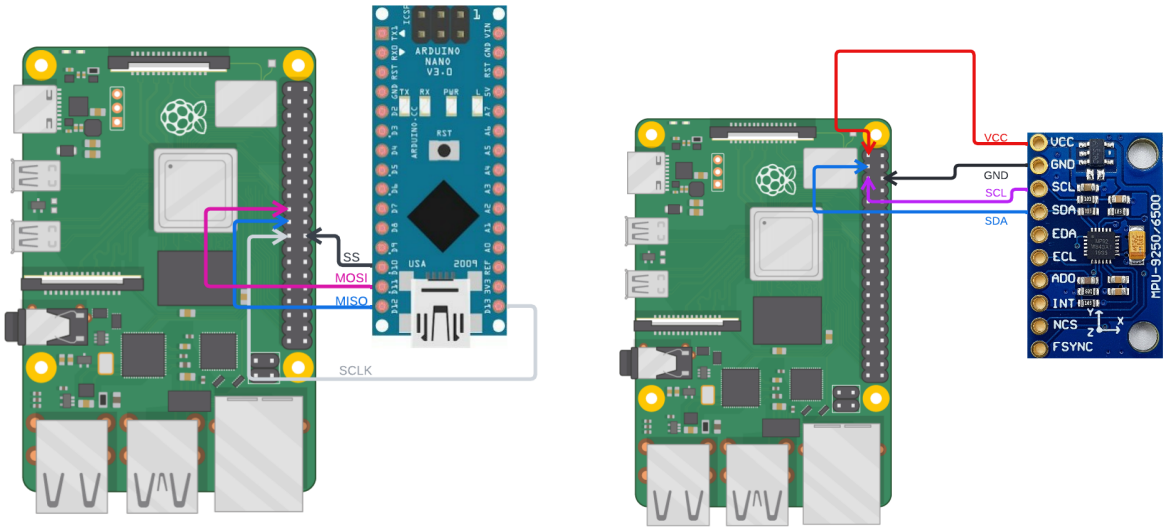


Figure 4.65: Connection diagram for the IMU, RPI, and Arduino

The code inspiration comes from the ElectroWings website [41]. This is using an MPU6050, and the only thing we need to change is the registers from the MPU9250 datasheet [57] we want to read and the SPI-channel together with the baud rate. To initialize the sensor, a function is created. This function writes to the registers needed to calibrate and set the accelerometer and gyro range, and also to power on the sensor. In the code, which can be viewed in G.9, this is called "MPU9250_init".

To read from the data registers, we use a function called "read_raw_data". This function demands one input, which is the address we want to read. The function then reads the most significant byte first, and then the least significant byte from the register. It then shits the most significant byte 8 bits to the left and adds the least significant bits into the first 8 bits. This means we will get 16-bits of data from the register. These bits are then assigned to a float variable in the main function and can be printed out using the printf command.

To initialize the SPI-functions in the Wiring Pi library, we do the same as mentioned in 4.11.2, but this time we open SPI-channel 0. To do a read/write operation we use the same function as before. This time we make a buffer pointer to hold the float value we want to send based on the Arduino example here [40]. This example is also used when creating the slave code for the Nano. Since the read/write function in Wiring Pi request a buffer of type uint8_t, we can create a uint8_t pointer that point to the float value. This is accepted into the function. This is shown in the code snippet below 4.18.

Listing 4.18: Mapping the float variable to uint8_t

```
float initial_value;

uint8_t *ptr = (uint8_t)&initial_value;

wiringPiSPIDataRW(SPI_CHANNEL, ptr, sizeof(ptr));
```

The slave side code can be viewed in G.10. The SPI and interrupt setup are the same as in G.7, but some new variables to hold the received data from the SPI are created. This is the same setup as in the master code, where a pointer of type uint8_t is pointing to the float variable, and when reading from the SPI data register, the data is stored in this pointer. We can then print the pointer using Serial.print(float variable), to print the data that was received.

This worked when sending one byte over the SPI-bus, but when sending two bytes, errors started occurring. This could be due to the fact that the pointer in the Arduino program was not able to hold all the data, and it was therefore tried to do two separate transactions from the master side. The first transaction contained the pitch, and the second the roll data. This worked to some extent, but due to the timing not being perfect, the values more often than not were printed as zeros. This was not tested properly, but it was strongly believed that the delay between each transaction on the master side have to be perfectly matched with the delay on the slave side. If not, there may be a transaction that only contains zeroes since the data still is being extracted on the master side or the slave side is not ready to read new data. Some interrupt routines on the master side may have to be implemented to avoid this issue. This must be evaluated further, but the end result of one-byte transactions can be viewed in the video. It is then confirmed that data transactions over SPI from RPI to Arduino are working, using the wiring pi library on SPI channel 0.

4.14 PID control and motor mixing

For stabilizing the drone, a PID control algorithm is the perfect implementation. This algorithm was initially intended to be used on the Raspberry Pi, as its processing power and speed are significantly better than the microcontroller. However, since there were complications reading the IMU data on the RPI, the PID algorithm was implemented into the Arduino simultaneously as working on the IMU code. This was done in case the IMU code worked on the RPI, and the drone could be flown by just extracting the IMU values, calculating the pitch and roll, and then sending it to the microcontroller.

4.14.1 Implementation

To set up the conditions to build a PID controller and test it, the MPU9250 was used again to provide the necessary data. The Arduino IDE contains a PID library, which was used in this test. The library is made by Brett Beauregard, which can be found here [20]. The library contains functions that make PID implementation easy and straightforward. To initialize the motors, the servo library from Arduino is used [7], together with a potentiometer to simulate the throttle. The final program can be viewed in G.11.

PID

To use a PID, we need input parameters for the controller and also a setpoint which we want it to regulate about. We need to provide the controller with the pitch and roll from the IMU, and then the PID can calculate the error according to the setpoint, and from there try to regulate the speed of the motors to acquire the setpoint.

We implement the MPU9250 with the Arduino library "MPU9250_WE" [135]. This contains a basic example of extracting data from the accelerometer and the gyroscope, and a program is made based on the example "MPU6500_all_data". Extract the acceleration data and use this to compute the pitch and roll based on the equations here: 2.5 2.6. The pitch and roll have their own PID controller, called myPID and myPID2 respectively. The PID function requires 7 inputs. These are the input data, output data variable, the setpoint, the three PID constants, and finally the direction 4.66.

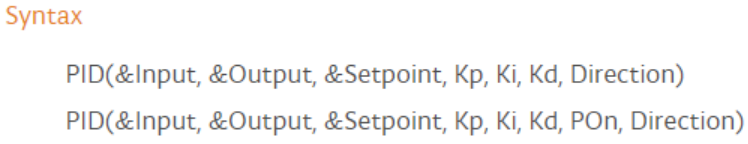


Figure 4.66: The PID function from the PID library [20]

For the PID control controlling the pitch, the input data is the calculated pitch from the IMU, and the output data is the calculated error that will go to the motor mixer. We set the setpoint to 0, as we want the drone to stay level at 0°. We do the same for the PID controller controlling the roll, substituting the pitch input with the roll input. In the void setup() function, we set the two PIDs to AUTOMATIC mode, as this automatically calculates the error based on the input parameters. In the void loop() function, we use the .Compute function to initialize the PID controllers. The output is then put into the input pulses for each motor in the motor mixer. The PID constants are initially set low, and when running the tests they are tweaked to get the fastest response. When tuned, the PID constants were: $K_p = 1$, $K_i = 0.01$, and the derivative constant was set to zero.

Though these values worked well during the test, the drone has yet to be flown, and the constants must be tuned thereafter. It is also worth mentioning that the two PID controllers are set to reverse, which means they only calculate the negative error. There should also be implemented a PID for calculating the positive error, with its direction being set to DIRECT. This can be seen in

the video [123], as turning the IMU towards negative pitch and roll values gives an error compensation from the PID, but positive pitch and roll give no output. One possibility, which was not tested, is to implement some form of loop or if-statement that tracks if the pitch and roll values are positive or negative, and based on this can calculate an output to write to the motors.

The sole reason the implementation of a PID on the Arduino Nano was done, was because the footprint for the ATmega was wrong on the second iteration of the PCB. The best possible way to implement a PID controller is to do it on the microprocessor, which in addition has a real-time operating system on it. The PID implementation in this case, was only done to be able to fly and test the drone if the IMU values could be read.

Motor mixing algorithm

To use the motors from the Arduino IDE, they have to be assigned to the Servo library. To simulate the throttle, a potentiometer is used. The potentiometer gives an out in the range of 0 to 1023, while the servo library only allows values from 0 to 180. A map function is used to convert the potentiometer output to a servo compatible value. The motors are assigned to the servo library by writing "Servo" followed by the motor name. The motors have been given names ranging from 1 to 4, where motors 1 and 2 are the right front and left back motors, and motors 3 and 4 are left front and right back. This is a basic layout for an x-frame quadcopter, which this drone is. The layout can be viewed in figure 4.67.

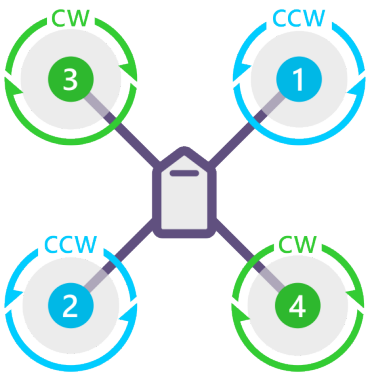


Figure 4.67: Quad X frame [9]

Then the motors should be assigned to the correct PWM pins on the Arduino. The Nano has 6 PWM pins, where four of them have a frequency of 490 Hz and the other two have 980 Hz. To avoid issues, the four motors are connected to pins 3, 6, 9, and 10 which have the same frequency. The range is set to 1000 <-> 2000 ms. The speed from the potentiometer is now combined with the outputs from the PID controllers. This is where the motor mixing happens. The pulse going into the motors have to be customized for each motor, depending on if it should speed up or slow down to keep the drone stabilized.

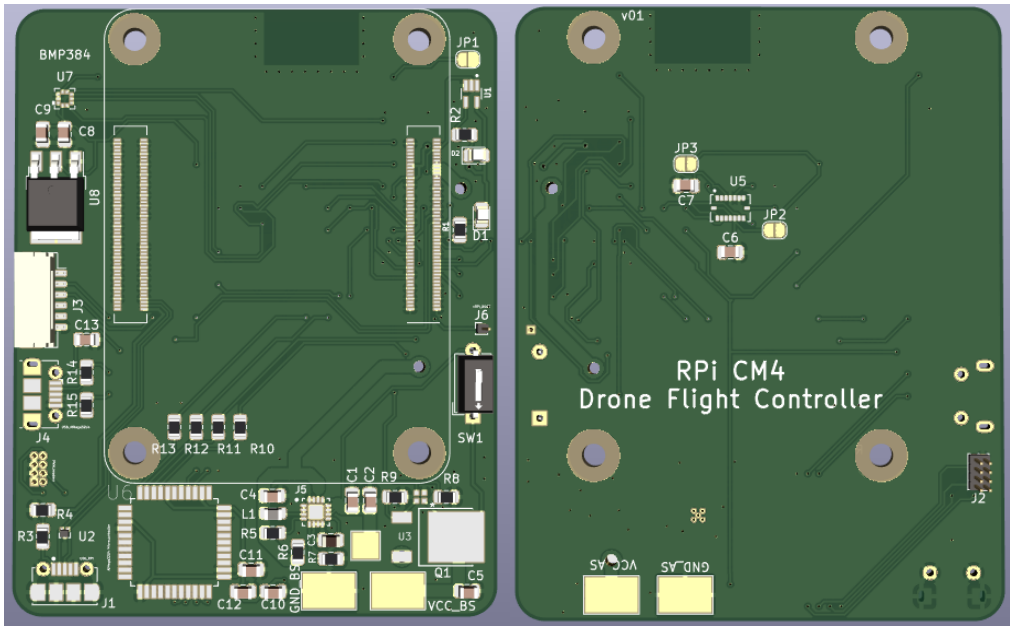
When the front of the drone is tilted downwards, the two front motors should speed up and the back motors should slow down to keep the drone level. Downwards tilt corresponds to a drastic change in pitch values. This means that motors 1 and 3 must have the PID value from the pitch error added to its speed. This is the opposite for motors 2 and 4, which needs this value subtracted. For upwards tilt, the roles would be reversed. When the drone is tilted to the right, motors 1 and 4 need to speed up and motors 2 and 3 must slow down to compensate. This means that motor 1 and 4 needs the PID output of the roll error added to them, and the opposite for motors 2 and 3. The finished motor mixing can be viewed in the code in Appendix G.11, and the pulses are written to the motors using the servo library write function.

Chapter 5

Results

5.1 PCB Design

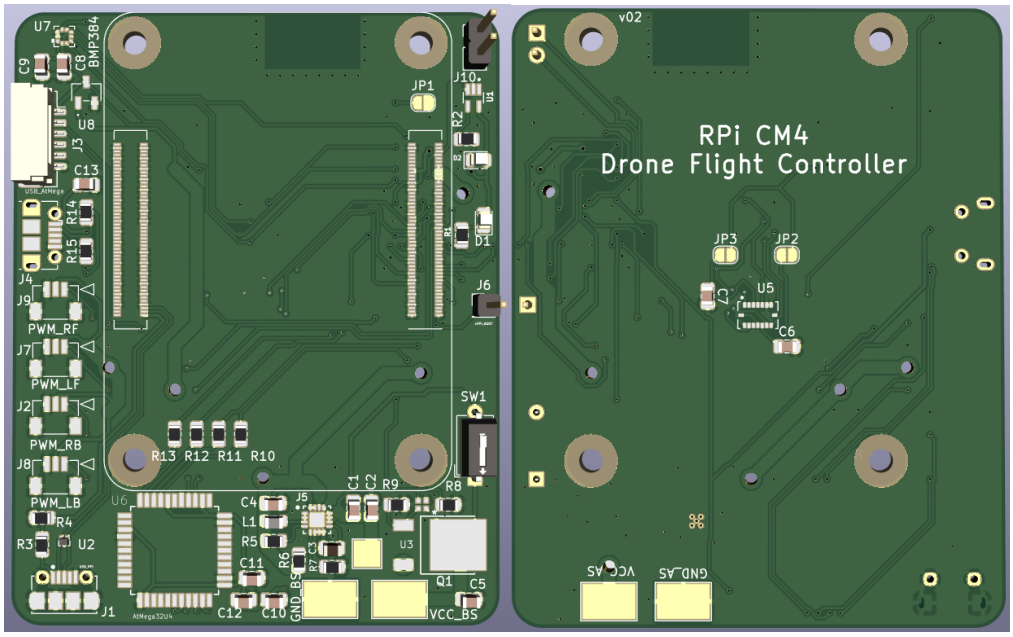
5.1.1 Version 1



(a) Front (b) Back

Figure 5.1: PCB version 1

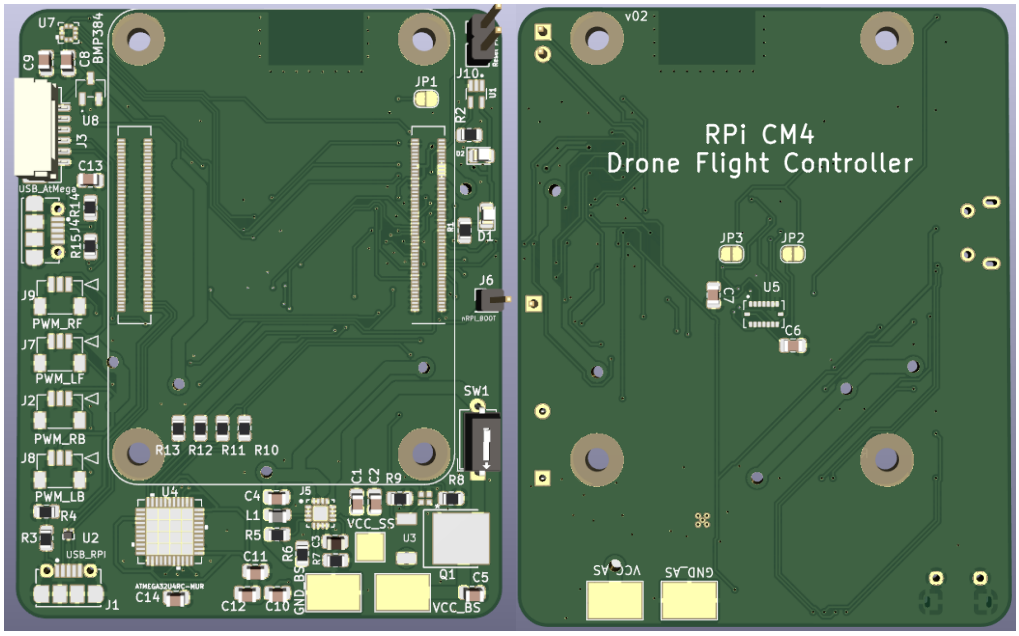
5.1.2 Version 2



(a) Front (b) Back

Figure 5.2: PCB version 2

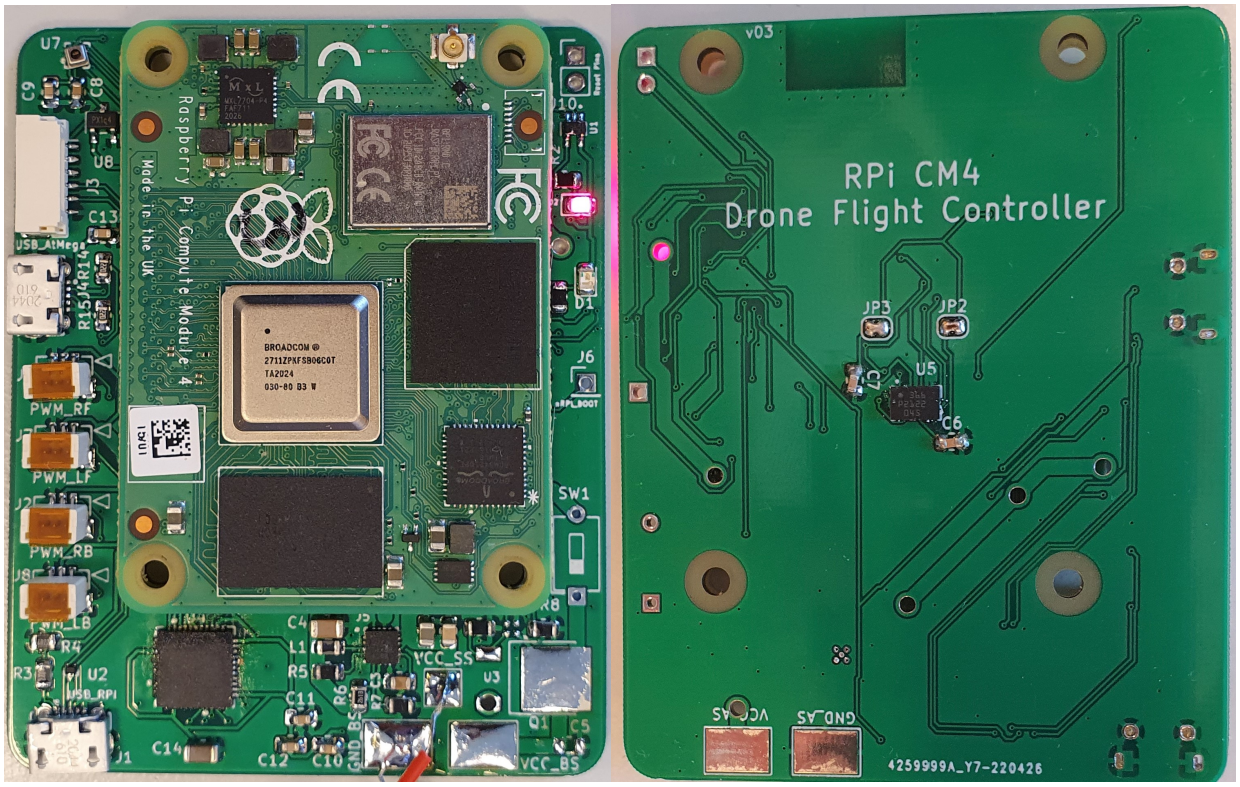
5.1.3 Version 3



(a) Front (b) Back

Figure 5.3: PCB version 3, Final version

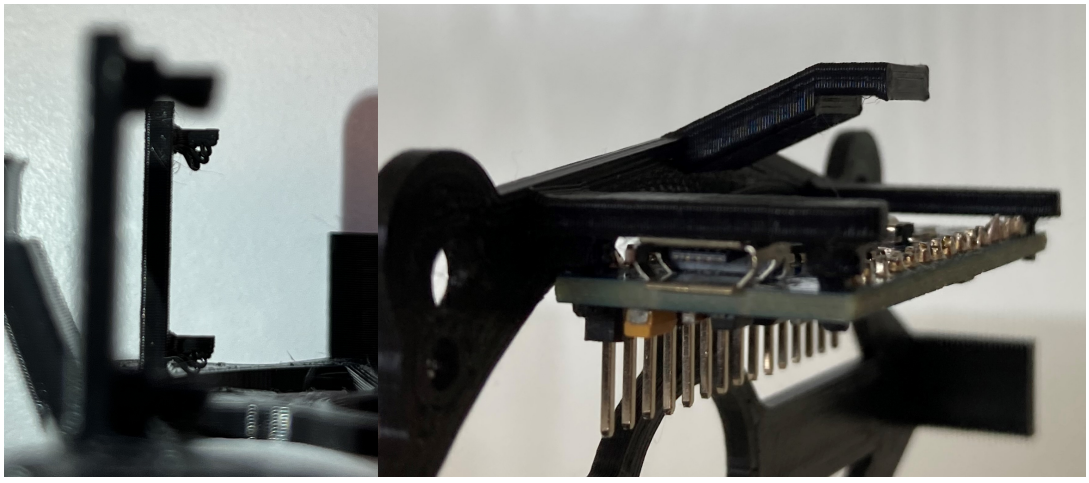
5.1.4 Fully built PCB



(a) Front (b) Back

Figure 5.4: PCB version 3 with all components attached, Final version

5.2 3D printed parts and mounting of parts



(a) As received from print (b) After removing excess filament and mounted Nano

Figure 5.5: PCB mounting bracket.

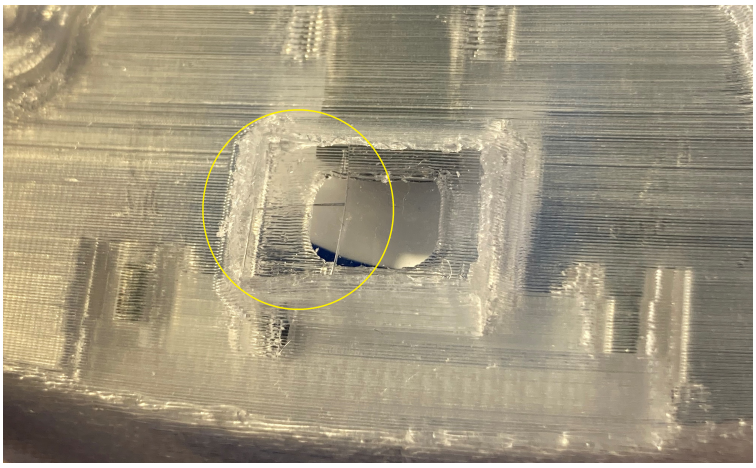


Figure 5.6: Lens cracked during an attempt to mount it.

The figure below, 5.7, contains one ToF sensor, lenses, ECSs, PDU, Motors, north, and south battery holder mounted to the bottom part of the drone shell.

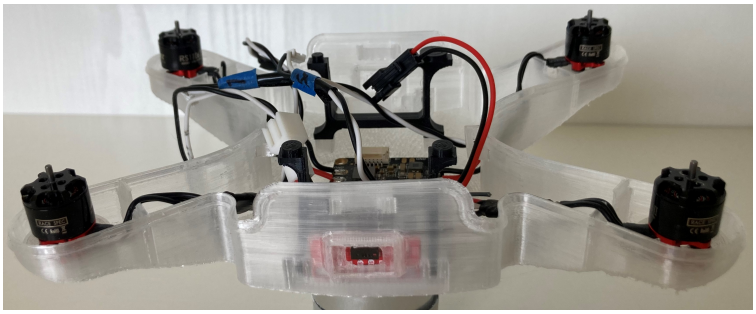


Figure 5.7: Bottom part of the drone shell with some parts mounted

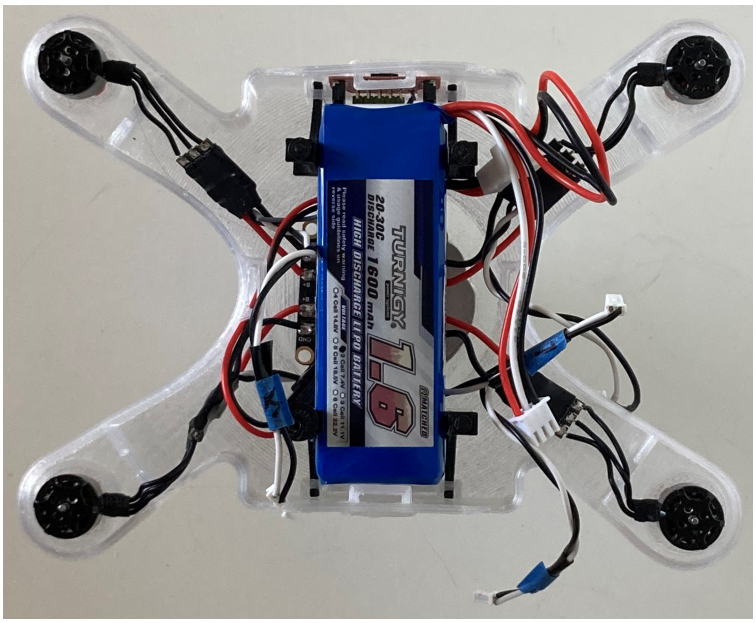


Figure 5.8: Same content as the figure above, 5.7, with the addition of a battery.

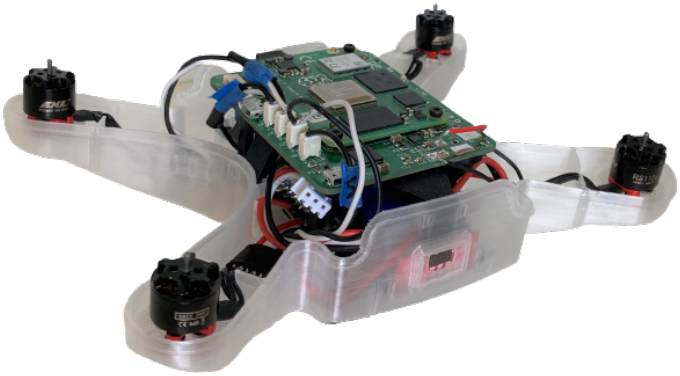


Figure 5.9: Assembled drone, without propeller and top

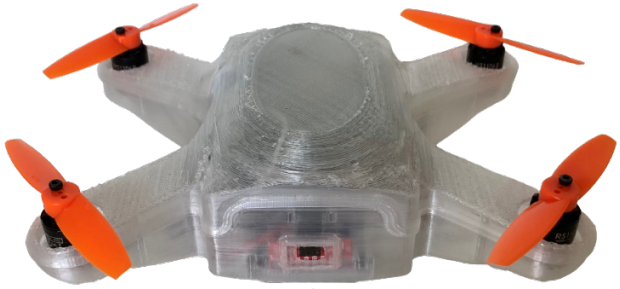


Figure 5.10: Assembled drone

5.3 ToF Sensor readings

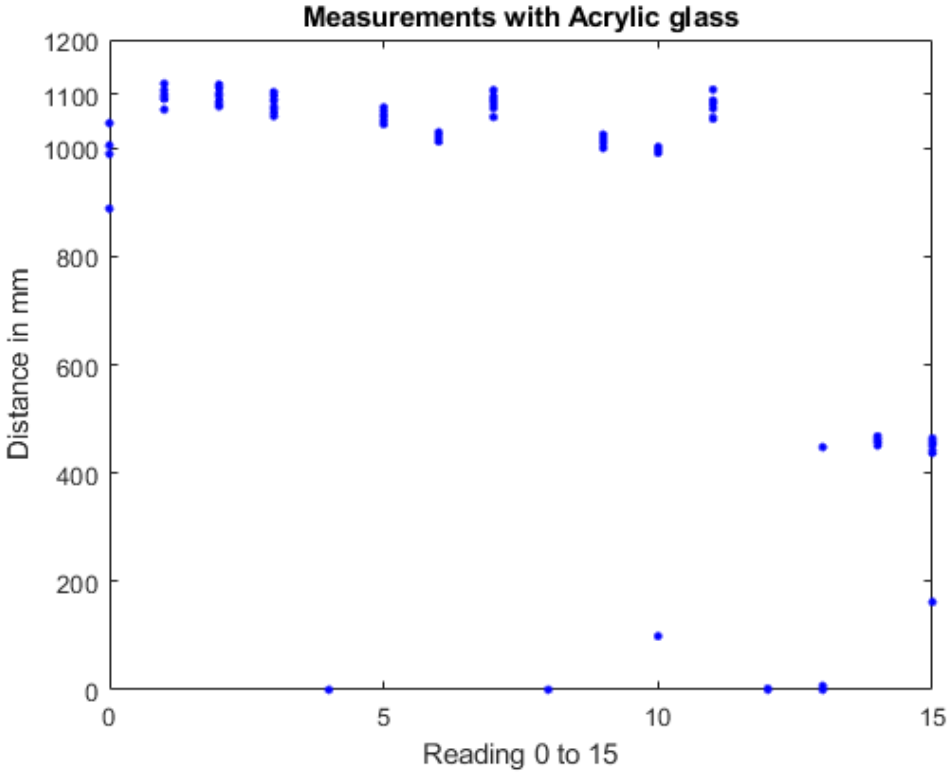


Figure 5.11: Measurements with the acrylic glass mounted

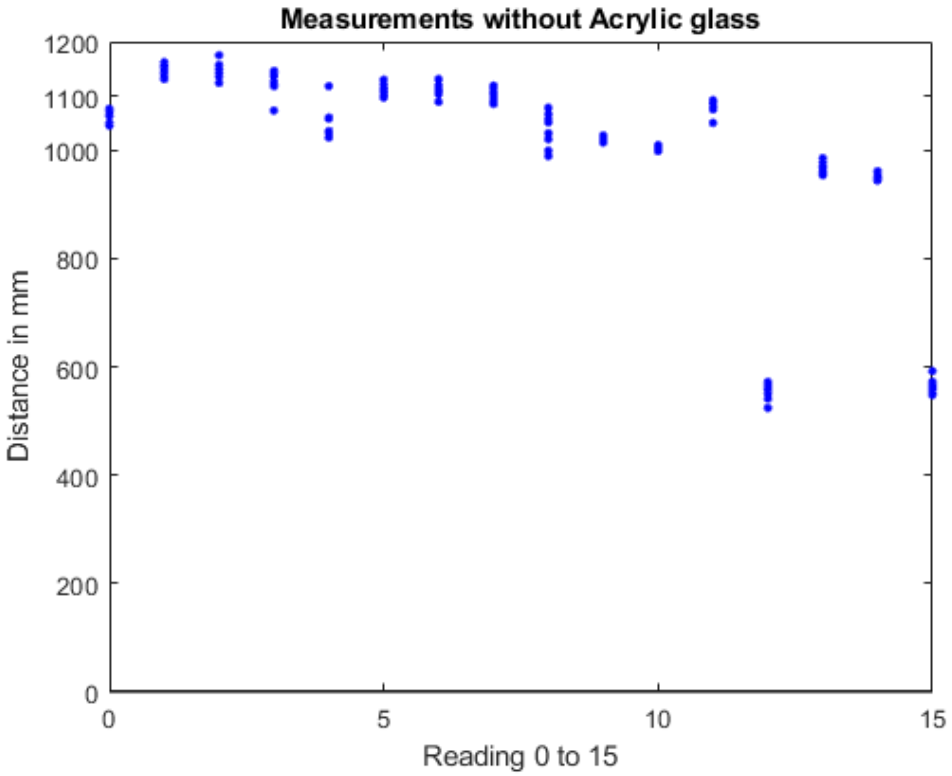
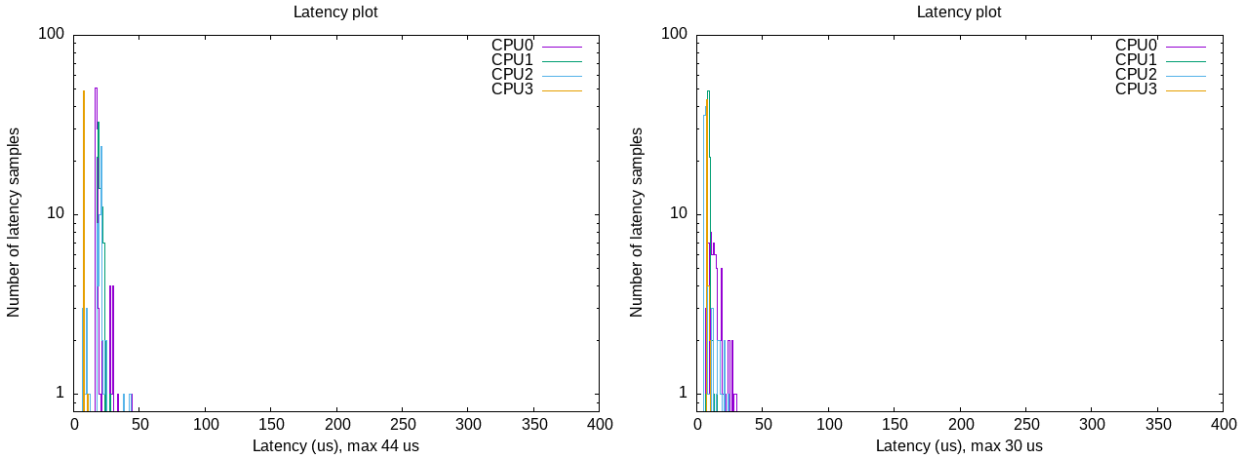


Figure 5.12: Measurements without the acrylic glass mounter

5.4 Real-Time vs Regular Kernel tests

5.4.1 Cyclictest



(a) 5.10.90 (b) 5.10.90-rt61

Figure 5.13: Regular kernel vs RT-kernel

```
# Total: 000000100 000000100 000000100 000000050
# Min Latencies: 00017 00018 00007 00008
# Avg Latencies: 00019 00019 00015 00008
# Max Latencies: 00044 00028 00043 00011
```

```
# Total: 000000100 000000100 000000100 000000050
# Min Latencies: 00007 00007 00006 00008
# Avg Latencies: 00012 00009 00008 00008
# Max Latencies: 00030 00015 00024 00010
```

Figure 5.14: Regular kernel vs RT-kernel | Table values

Ptsematest

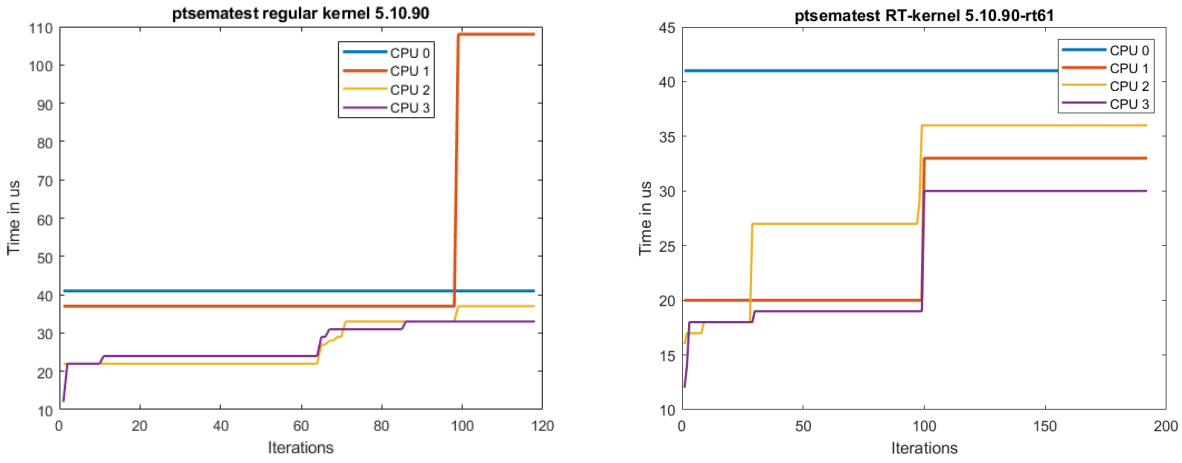


Figure 5.15: Plots of the test. Regular vs Real-Time

Migrate test

Table 5.1: Migrate test | Regular Kernel 5.10.90

	Task 0(prio 2)	Task 1(prio 3)	Task 2(prio 4)	Task 3(prio 5)	Task 4(prio 6)
Max	20144 us	142 us	132 us	118 us	138 us
Min	46 us	46 us	45 us	43 us	47 us
Tot	943799 us	3696 us	3302 us	3207 us	3207 us
Avg	18875 us	73 us	66 us	64 us	64 us

Table 5.2: Migrate test | RT-kernel 5.10.90-rt61

	Task 0(prio 2)	Task 1(prio 3)	Task 2(prio 4)	Task 3(prio 5)	Task 4(prio 6)
Max	20146 us	207 us	108 us	101 us	115 us
Min	60 us	50 us	46 us	43 us	46 us
Tot	445127 us	5052 us	3354 us	3322 us	3514 us
Avg	8902 us	101 us	67 us	66 us	70 us

Sigwaittest

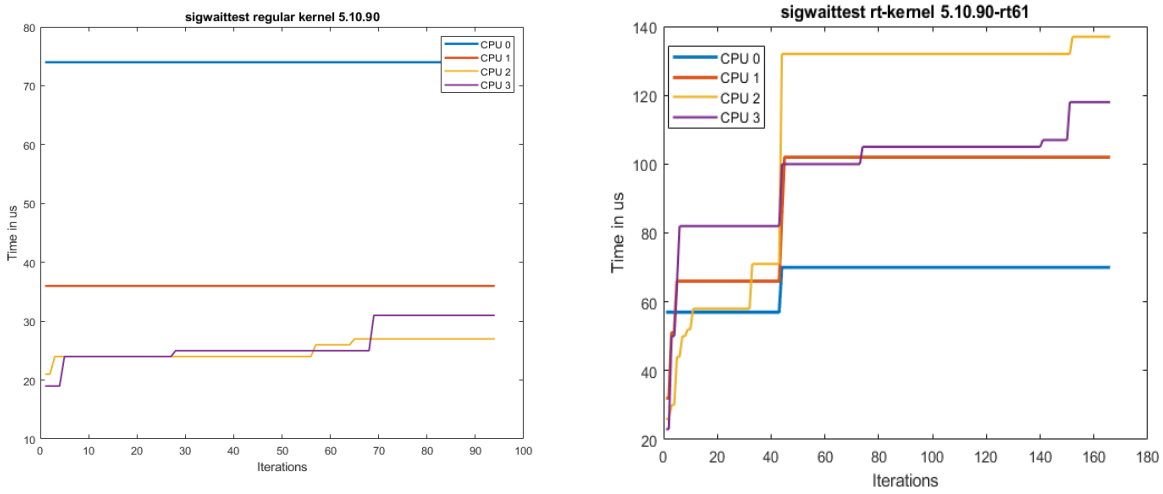


Figure 5.16: Plots of the sigwaittest. Regular vs Real-Time

Svsematest

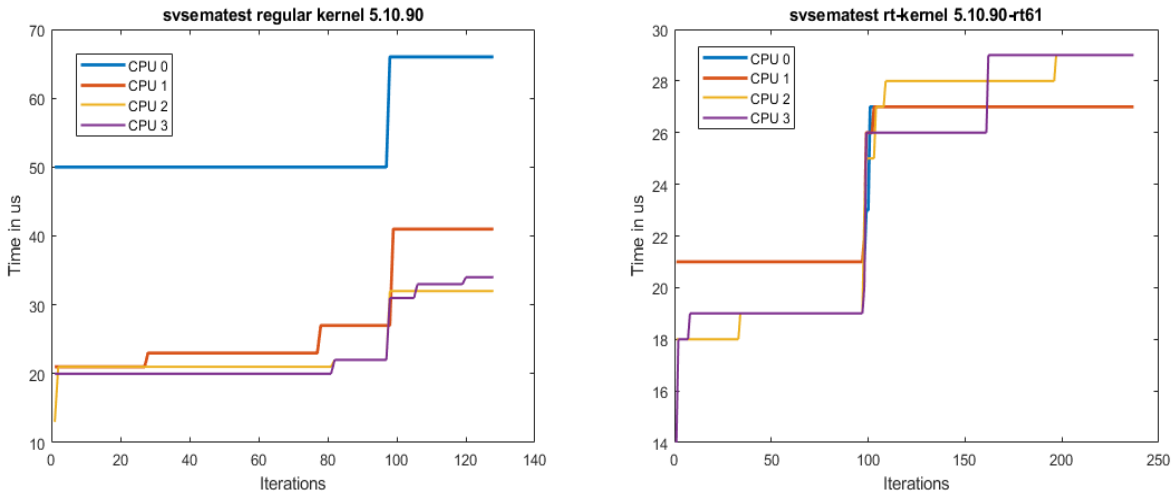


Figure 5.17: Plots of the svsematest. Regular vs Real-Time

N-Queens Problem

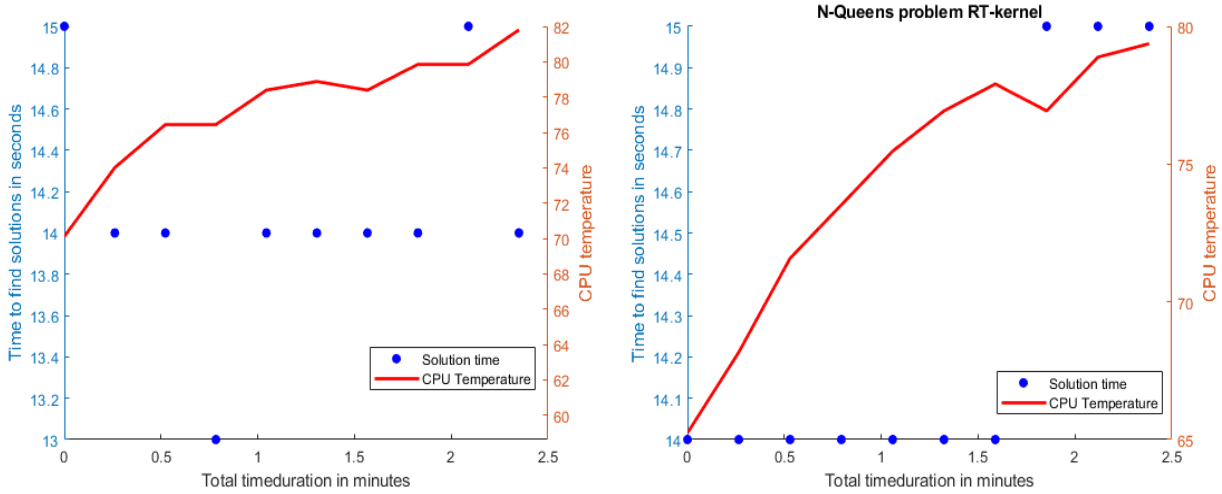


Figure 5.18: N-Queens problem Regular vs Real-Time

Table 5.3: Data contents of N-Queens problem on 5.10.90

time	queen	threads	solutions	seconds	microseconds	cpu_temp	gpu_temp
1648632701	12	4	14200	15	467173	70.11	71.50
1648632715	12	4	14200	14	137553	74.01	74.00
1648632729	12	4	14200	14	369621	76.44	76.40
1648632744	12	4	14200	13	985760	76.44	76.40
1648632758	12	4	14200	14	314642	78.39	78.40
1648632773	12	4	14200	14	811183	78.88	78.80
1648632787	12	4	14200	14	678306	78.39	78.40
1648632802	12	4	14200	14	959134	79.85	79.80
1648632818	12	4	14200	15	372213	79.85	79.30
1648632832	12	4	14200	14	608437	81.80	81.30

Table 5.4: Data contents of N-Queens problem on 5.10.90-rt61

time	queen	threads	solutions	seconds	microseconds	cpu_temp	gpu_temp
1648633615	12	4	14200	14	603514	65.24	65.20
1648633629	12	4	14200	14	164890	68.17	69.60
1648633644	12	4	14200	14	225153	71.58	71.50
1648633658	12	4	14200	14	466564	73.52	73.50
1648633673	12	4	14200	14	924247	75.47	75.40
1648633688	12	4	14200	14	676450	76.93	76.40
1648633702	12	4	14200	14	441396	77.91	77.90
1648633717	12	4	14200	15	176947	76.93	76.90
1648633733	12	4	14200	15	348695	78.88	78.80
1648633748	12	4	14200	15	379792	79.37	79.30

Chapter 6

Discussions

6.1 Global Electronic Component Shortage

The global component shortages have gotten worse after the recent Covid-19 lockdown in most of the big cities in China. The lockdown has affected the big microchip factories which affect everything from car manufacturers to phone manufacturers all over the world. This is relevant because of the need for small electronic components and microchips for this project. The project was affected by this in the terms of component selections. For example, was it originally thought to use the BMI088 IMU, but because of the shortage was not this possible (Section 4.4.4).

6.2 PCB Design and build

During the design and mounting of components on the PCB, multiple problems arose. Some were solved and some led to a discussion on the future improvement of the PCB to enhance quality, modularity, and debugging options. To avoid any misinterpretations the V0 of the PCB were made during a previous project [79] and V1, V2, and V3 were made during this project.

6.2.1 Solder paste

During the first attempt to attach the components to the PCB (Figure 4.27) some problems occurred. The solder paste did not flow as easily through the dispenser needle as it should, which led to the use of a bigger needle that was not small enough to hit the small connection pads on the PCB properly. It turned out that the solder paste was expired and therefore was not as fluent as it was meant to be. The problem was fixed by ordering a new solder paste (Figure H.1). The problem was not attended to as soon as it should have been, because there was a insecurity concerning if it was the paste or the skill level of the person soldering who was the problem. This was the first time using solder paste for all of us.

6.2.2 HIROSE Connector

When testing each PCB (Section 4.7.4) with RPi CM4 after the soldering process some problems emerged. The CM4 did not boot properly which led to no WiFi connection. The problem was that since the pins for attaching the HIROSE connector were as small as they were, some solder had found its way over to another pin which led to a short-circuit of the connector as shown in (Figure 4.32). The problem was then fixed by heating the solder and using a desoldering braid to remove the leftover solder.

The upside of the size of the HIROSE connector was that it gives a large amount of connectivity but takes almost no space. This was good when considering the space limitations to the flight controller in this project, as it was to be mounted inside a drone. If mounted correctly it also makes taking the RPi CM4 on and off easy. Since the RPi CM4 multiple times was moved between the carrier board and IO board this was a good function to have.

The downside with the HIROSE connector was the extreme sensitivity on mounting it right. Given the incredibly small size of the pins and pads, it was very difficult to ensure there were no possible short-circuit between any of the pins. It was checked under a microscope to see if we could see any overlap between the pins, but due to the plastic casing on the connector laying over most of the pin and the pad, it became almost invisible. The worst and biggest overlaps were visible. Still, after checking for it, the HIROSE did not always work, which often resulted in the RPi CM4 not booting. The problem was that if it was even a small overlap between the pins, it could result in the RPi not booting. An other result could be that the RPi booted, but it still had a overlap on the GPIO pins, which led to other problems.

This was hard to debug, as it was not visible, and might just give no signals when trying to send or receive. This was problematic when using sensors that relied on multiple signals from the GPIO pins, such as the IMU connected with SPI. It has therefore been hard debugging connection problems with IMU and barometer via SPI. The carrier board is small and the HIROSE even smaller, so even though it might be a good solution if fitted correctly, it might be an idea to look at other solutions during the testing phases of a project. Another solution might be to order the PCB from the manufacturer with the HIROSE already mounted, as they are very likely to endure two times in a reflow oven, therefore they can be mounted first from the manufacturer, and then the rest of the components can be mounted manually later.

6.2.3 Small components

There were some problems with the smallest components. The barometer was not easy to attach properly. It had a tendency to move during the reflow process, which led to a misplacement of the component on the PCB and as a result, the pads on the barometer did not connect with the right pads, thereby giving no connection. If we were to attach all the components again on a new iteration, it would be easier to attach the barometer after the reflow process in the oven, by attaching it by hand with some solder paste and a hot air rework station.

6.2.4 Voltage regulator

The battery used in the drone for this project gives out 7.4V, Which is more than the optimal voltage for the rest of the PCB. So one problem that occurred was that during the reflow process, the voltage regulator got short-circuited by too much solder paste, and therefore did not regulate the voltage down to 5V. This led to that 7.4V being put directly into the LDO (Section 2.1.3) and fried the component which made the iteration of the board useless. This was solved by going over each component on the next iteration of the PCB under a microscope to make sure that there was not any extra solder paste that could potentially ruin more components by short-circuiting them.

6.2.5 Attaching components by hand

The IMU was placed on the backside of the PCB (Section 4.4.4) which made it hard to use the reflow oven due to the components on the front side of the PCB that were not recommended to go twice in the oven. This meant that all the components on the backside of the card as shown in (Figure 4.6) had to be done by hand. The bigger components as for example the capacitors were not any problem, but the IMU needed some practice runs with the heating gun and some solder paste to be properly fastened to its solder pads. If we were to send yet another iteration of the PCB to be manufactured, it could be an idea to look for an IMU with legs as it would be easier to get a good result with a soldering iron.

6.2.6 Stencil

On our last iteration of mounting the components to the PCB V3 (Section 5.3) it was decided to order the optional solder stencil to try to make the attachment of the components easier. It became a lot easier to get solder paste to each of the small solder pads, but it was still necessary to go over and remove some of it later. If the solder stencil would have been ordered with each

iteration of the PCB, it would probably have resulted in fewer mistakes, as shown in figures 4.26, 4.27, 4.28, 4.29, 4.30, and 4.31.

6.3 Mechanical drone development

6.3.1 3D printed parts

Regarding the results from the 3D printed parts. They functioned as designed, but needed some physical work to function optimally. Some altercations occurred when fitting the parts together and therefore led to discussions on future improvement (Section 8.2).

6.3.2 Lens holder

The lens holder worked as intended, the lens was unable to come loose unless intended. The fitment was tight, which led to one of the lenses cracking during an attempt to mount it, see figure 5.6. One is able to mount the lens by hand. However, it is found by using a flat head screwdriver to "shoehorn" the lens in place, makes the process of mounting the lens substantially easier. It is undetermined if shrinking the lens, or expanding the lens holder, will make mounting easier. And yet still keep the lens flush with the body and not fall out under an impact.

6.3.3 PCB mounting bracket

The printer struggled to make the pins for the Arduino Nano, see figure (a) 5.5. This was due to it being difficult for the printer to make such a thin support, which then again made the overhang too big for a good result. To place the Nano excess filament was removed manually. Further development for Arduino mounting was not pursued due to its low use case and not being fitted in future generations of UiA's drone project.

After having printed multiple iterations of the PCB mounting bracket the end result worked as suspected. By just altering the bracket from the Parallel project, [61], and not a designing a one from scratch. It sat limitations during development so in retrospect, it would be more beneficial to design one from scratch, and just use the same anchor point. Then it would easier to use the PCB stack as a structural member, rather than mounting it with little to no structural properties, as it has now. An additional improvement could be to relocate the two northern holes, for the battery holder. This would make the PCB stack assembly easier.

6.3.4 North and south battery holder

The battery holders gave the bottom part of the drone great stiffness. It held the battery in place with little movement. The northeast spacing left for the nut from the PCB stack bolt was adequate to fit it. However left very little room for mounting, and we had to use a special tweezer to hold the nut still, while fitting it onto the bolt. By altering where the northern battery holder mounts to the mounting bracket this could have been circumvented, as mentioned in the subsection above, 6.3.3.

6.3.5 ToF sensor placement result

After struggling with having any connections trough the Qwiic JST connector, our hypothesis was because it had too long cables during testing, was not looked further at. We ended up using the pinouts on the shuttle board instead. The ToF sensor was just for testing the use of acrylic glass vs. without and not configured to be in use under flight. It can be seen from figure 5.8, that there is very little empty space between battery and the bottom shell. This meant that the drone was not able to fit two ToF sensors. Therefore it was only fitted to one side. This was due to the connectors on the shuttle board taking up to much space. A possible solution to mount more than one ToF sensor is described in the Further development section 8.2.

6.3.6 Discoveries from the drone assembly

The building instructions in Appendix J showed that the PCB stack got mounted after the mounting bracket was in place. The fact was that it was quite cumbersome to mount the stack to the bracket, due to the low clearance between the battery and bracket. So for an easier mounting process, it is recommended to mount the stack to the bracket before putting the PCB mounting bracket in place. The four holes made for each motor mount should also be rotated about 90 degrees either way, from the motors center. This is because in this iteration, the wires coming from the motor do have to be bent quite hard to fit the entry slots (Figure 4.36).

6.4 ToF sensor test result

The results from testing the ToF sensor with and without acrylic glass gave us an indication of how the ToF sensor reacts to having an obstructed view. By looking at the differences in results illustrated in figures 5.12 and 5.11, both the similarities and dissimilarities are clear. For example, the values from reading 9, 10, and 11 are close to identical for both tests. However, the test with the acrylic glass has the reading of 4, 8, and 12 equalling zero. This can not be considered reliable data, further testing is recommended to find the exact reason for these low values.

6.5 Real-Time Operating System

The need for an RTOS in a drone setting is, as discovered, highly necessary to ensure a responsive and stable drone.

6.5.1 Test results

In figure 5.13 the results of the cyclictest can be viewed. Here, we can see that the RT-kernel outperforms the regular kernel by a 14 μs difference in the maximum measured latency. This means that the RT-kernel is about 1.5 times faster than the regular kernel. Ideally, this test should be run for a much longer time than what was done in this project. The load over time would give us the overall performance, so this is something which we wish we did. The test nonetheless gave us an indication that switching to the RT-kernel was a good choice, as any kind of latency minimization is to the drone's advantage when operating.

The ptsematest also shows us that the RT-kernel outperforms the regular one. The test measures the latency between sending and getting locks, and as shown in plot 5.15 the latency measured in each core of the CPU stays pretty consistent over the number of iterations. The CPU cores 0, 2, and 3 are quite similar to the regular kernel to the RT-kernel, but the latency of the CPU 1 increases drastically when exceeding 100 iterations on the regular kernel. In the RT-kernel on the other hand, the latency of CPU 1 stays low. It is clearly seen that the first core is executed first each time, since it has the highest priority setting, with an average latency of a bit over 40 μs, and the other cores are executed after this as the lock is acquired. The other cores latencies are also very low, which tells us that the kernel reacts fast to the lower priority cores when the first one is executed.

The migrate test gives an indication if the task scheduler in the kernel in fact executes the highest priority tasks first. In tables 5.1 and 5.2 the results are presented, and the most interesting part of the test is the maximum measured latency. As expected, the RT-kernel also outperforms the regular kernel, though not by as much as we believed it would. Task 4 is the task with the highest priority, and the difference between the kernels is 23 μs. Although this is a good outcome for the RT-kernel, the maximum latency is still above 100 μs. This is 70 μs larger than measured in the cyclictest, so this was quite peculiar at first sight. A note here is that the cyclictest only measured the latency of one thread with the highest priority on each CPU core. As for the migrate test, it tests the latency between 5 different tasks with different priorities. It then makes sense that the maximum value would be larger than in the cyclictest. The regular kernel also has the best minimum latency data, but the RT-kernel still wins by having the lowest maximum latency. Overall the kernels were quite similar in this test.

The sigwaittest however gave some interesting results. As seen in the plots in 5.16, the latency of the regular kernel severely outperforms the RT-kernel. The y-axis can be a bit deceiving, but nonetheless we can see that the latency of CPU 1, 2, and 3 in the RT-kernel are much larger than for the regular kernel. The first CPU core stays the same, as this has the highest priority. At least the kernels execute this task first, and they are both similar in the plots. The lower priorities are executed much faster on the regular kernel, but as one can see, they do not have the same iteration cycles. This is a user error, as the duration term was not used. This test should be redone with the same duration to obtain a proper comparison, as there may be a chance the regular kernel latencies would have been much different had it also been run for 180 cycles.

The svsematest was also ran with different iterations, which again is a user error. Nonetheless, the plots in 5.17 show that the RT-kernel now outperforms the regular kernel drastically. This test is very similar to the sigwaittest, but the svsematest utilizes SYSV semaphores, as mentioned in 4.9.2. The results indicate that the RT-kernel utilizes semaphores much better than the regular kernel, and due to this makes a superior system. This can also justify the results in the sigwaittest, where semaphores were not used. This tells us that the use of semaphores is very

important for the RT-kernels overall functionality and this must be looked at how to implement properly when the drone is finished.

The final test, which was the N-Queens test, was inspired by LeMaRivas RT-test[76]. LeMaRivas result was that the RT-kernel was slower at solving the maths problem, so we wanted to test if this was the case for our RT-kernel as well. In the tables in 5.18 the results are shown, and same as LeMaRiva, the RT-kernel was slower, though not by much. The regular kernel averaged a solution time of 14.1 seconds, whereas the RT-kernel averaged a 14.3 second solution time. This difference is not that big, and the CPU temperature was almost exactly the same for both kernels. Solving a maths problem is not the main purpose of making a kernel preemptible, but it is good to know that the performance on this aspect is not much different from a regular kernel.

6.5.2 Evaluation of the Linux RT-patch

Considering all the tests above, it is safe to say that making the Linux kernel preemptible improves the system's overall response time and the latency is much lower in many aspects. This is a good environment for a drone to operate in, and since it requires fairly easy steps to implement, it is one of the best choices for the CM4 in this project.

6.6 Frameworks

A framework is highly appreciated when working with embedded systems such as these drones. Having a support frame which provides a basic API makes the coding process much quicker. Whether this should be F' or not, is further discussed in this chapter.

6.6.1 Component implementation in the framework

As mentioned in 4.10.4, there were some issues implementing components into the framework. The RPi demo worked fine, and by the words of the makers of F', the best way to further develop an implementation on a Raspberry Pi, is to use this demo. The first test was done to see if the implementation of a barometer was easy to do or not. The GPS tutorial was the main influence and reference for making the code. It is stated in the tutorial that it is under revision for testing with F' version 2.0.0, but since we use the latest version (version 3.0.0), we thought this was sorted out by the developers. When following the tutorial, the first sequence of making the ComponentAi file went pretty well, but as soon as we tried to build the implementation files, several errors occurred.

Firstly, the build utility in F' was not able to locate some of the drivers we wanted to use. The GPS tutorial is based on serial communication, but we wanted to use I2C to access the MS5611 barometer. The compiler was able to locate the I2C drivers but was not able to locate the port Ai's from the framework. This was rather odd, as the tutorial used these, but the error message stated that these files do not exist.

As seen in figure 6.1, the builder fails to find the command port file. We then searched the directory where this file was supposed to be, but could not find it. This was very odd since the tutorial specified this, and we then thought that maybe the repository was not up to date. We cloned the repository one more time and did the same procedure. The first try was without luck, but the second try worked without any errors. We had not changed anything, and this led us to believe that the build functions of F' might not be that good. It is also very hard to debug this, as it worked almost out of the blue. The build utilities might then be considered unreliable, and this was the first indication that this framework might not be a good suit for our application.

```
student@jakob-VirtualBox:~/fprime/BarometerApp/Barometer$ fprime-util impl
Scanning dependencies of target codegen
[100%] Built target codegen
Scanning dependencies of target BarometerApp_Barometer_impl
[100%] Generating ../../../Barometer/BarometerComponentImpl.hpp-template, ../../
../Barometer/BarometerComponentImpl.cpp-template
Parsing Component barometer
Reading external dictionary /home/student/fprime/BarometerApp/Barometer/Commands
.xml
Reading external dictionary /home/student/fprime/BarometerApp/Barometer/Telemetr
y.xml
Reading external dictionary /home/student/fprime/BarometerApp/Barometer/Events.x
ml
ERROR: Port xml specification file Fw/Cmd/CmdPortAi.xml does not exist!
make[3]: *** [BarometerApp/Barometer/CMakeFiles/BarometerApp_Barometer_impl.dir/
build.make:62: ../Barometer/BarometerComponentImpl.hpp-template] Error 255
make[2]: *** [CMakeFiles/Makefile2:7703: BarometerApp/Barometer/CMakeFiles/Barom
eterApp_Barometer_impl.dir/all] Error 2
make[1]: *** [CMakeFiles/Makefile2:7710: BarometerApp/Barometer/CMakeFiles/Barom
eterApp_Barometer_impl.dir/rule] Error 2
make: *** [Makefile:2133: BarometerApp_Barometer_impl] Error 2
[ERROR] CMake erred with return code 2
```

Figure 6.1: Error message implementing Ai files

We continued trying to build the component, and the rest of the tutorial worked. Or so we thought. When launching the GUI, the commands we had specified in the Command.xml file was not present. This was because they had not been implemented into the RPi topology file, so when the build process is done, the barometer app is not included at all. In the method chapter 4.10.4, it was mentioned that to launch the GUI, the "RPITopologiAppDictionary.xml" has to be included. Since the barometer is not implemented in this, it is not weird that the commands did not show up in the GUI.

We then tried to implement our files into this xml file. There was no guide on how to do this, and not being experienced with the framework, this process was very hard. We tried to decrypt the dictionary file, to find out where we should implement the barometer. The file contained over 1000 lines of code, none of which gave a direct hint to where to put our files to build it together with the RPi demo. As the PCB arrived and we got the sensors attached, the more pressing matter was to make programs that could read these sensors, and if successful, we would try to implement them in F'. As this report has shown, getting those data was quite difficult, and no further testing of the framework was done. We however obtained a general opinion on how the framework worked, and because of the lack of guiding and complicated interface of the framework, we decided based on this that ROS would be a much better alternative.

```
Traceback (most recent call last):
  File "/home/jakob/fprime/cmake/autocoder/ai-parser/ai_parser.py", line 209, in <module>
    main()
  File "/home/jakob/fprime/cmake/autocoder/ai-parser/ai_parser.py", line 100, in main
    print_fprime_dependencies(root, *sys.argv[2:])
  File "/home/jakob/fprime/cmake/autocoder/ai-parser/ai_parser.py", line 129, in print_fprime_depend
encies
    dependencies = read_xml_file(root, import_base)
  File "/home/jakob/fprime/cmake/autocoder/ai-parser/ai_parser.py", line 200, in read_xml_file
    tmptree = xml.etree.ElementTree.parse(
  File "/usr/lib/python3.8/xml/etree/ElementTree.py", line 1202, in parse
    tree.parse(source, parser)
  File "/usr/lib/python3.8/xml/etree/ElementTree.py", line 584, in parse
    source = open(source, "rb")
FileNotFoundError: [Errno 2] No such file or directory: '/home/jakob/fprime/GpsApp/Gps/Commands.xml'
CMake Error at /home/jakob/fprime/cmake/autocoder/ai_xml.cmake:132 (message):
-- Configuring incomplete, errors occurred!
  Failed to parse
See also "/home/jakob/fprime/BarometerApp/build-fprime-automatic-native/CMakeFiles/CMakeOutput.log".
  /home/jakob/fprime/BarometerApp/Barometer/BarometerComponentAi.xml.  1
See also "/home/jakob/fprime/BarometerApp/build-fprime-automatic-native/CMakeFiles/CMakeError.log".
Call Stack (most recent call first):
  /home/jakob/fprime/cmake/autocoder/ai_xml.cmake:68 (__ai_info)
  build-fprime-automatic-native/cmake_language.cmake:388 (ai_xml_get_generated_files)
  /home/jakob/fprime/cmake/autocoder/autocoder.cmake:177 (cmake_language)
  /home/jakob/fprime/cmake/autocoder/autocoder.cmake:95 (__ac_process_sources)
  /home/jakob/fprime/cmake/autocoder/autocoder.cmake:37 (run_ac)
  /home/jakob/fprime/cmake/target/build.cmake:117 (run_ac_set)
  build-fprime-automatic-native/cmake_language.cmake:72 (build_add_module_target)
  /home/jakob/fprime/cmake/target/target.cmake:93 (cmake_language)
  /home/jakob/fprime/cmake/target/target.cmake:132 (setup_single_target)
  /home/jakob/fprime/cmake/module.cmake:40 (setup_module_targets)
  /home/jakob/fprime/cmake/module.cmake:80 (generate_base_module_properties)
  /home/jakob/fprime/cmake/API.cmake:162 (generate_library)
  Barometer/CMakeLists.txt:6 (register_fprime_module)


[ERROR] CMake erred with return code 1. Partial build cache remains. Run purge to clean-up.
jakob@jakob-VirtualBox:~/fprime/BarometerApp$
```

Figure 6.2: Error message trying to implement the component into the dictionary file

6.6.2 F' vs ROS

Previously, the UiA drones have used the Robotic Operating System, also known as ROS. ROS commonly operates using nodes. These nodes are published onto a network where they can be called to by other nodes. Some nodes are listeners, and some are publishers or both. In the drone aspect, this could be the IMU being a publisher node and publishing its data to the network. Other nodes can then subscribe to the IMU node and obtain its data. This could be a PID node, which can use the IMU data to control the drone motor speed.

In F' one builds components and implements them into a project. The project is then built using the built-in fprime-tools, and from there the user can cross-compile the binary executable file to a desired platform. The components are also imported into the GUI, which gives a live view of the channels and events of the deployment, and with the possibility of executing commands set by the user.

One can compare the use of nodes in ROS to the use of components in F'. It is quite similar considering the fact that a ROS node is nothing but an executable file containing ROS packages as explained by the ROS Organization [96]. The F' components are also files containing source code for the component and the framework packages. They both operate on a network and can be used with many different interfaces and operating systems. The only difference is that F' has a GUI directly implemented into the framework, as for ROS you would have to install these. The F' framework also automatically generates code for the components as a built-in tool. This could make it easier for the user to implement the components without having to write all the code by themselves. In ROS, one would have to do this. There are pros and cons to this, as writing the code independently would ensure that the user gets the exact output they want, as auto-generated code might not suit their coding style or be at all desired in their application. This report has also shown that implementing basic sensors in the framework is not straightforward, and ROS provides a much better interface for sensor implementation. ROS is used in the three previous bachelor projects and has proven to work well with drones [62] [72] [79].

6.7 Sensor interfaces

The interface used on the PCB is, as explained, the SPI protocol. The sensor we used, BMI085, has both I2C and SPI protocols and since SPI is the fastest, this was decided as the interface to use. Setting up code to read SPI values was something new to us, but when initially testing the SPI connection between the Raspberry Pi and Arduino Nano, it seemed fairly manageable. Later it was discovered that this protocol is not as straightforward as the I2C protocol, which we are familiar with. There were ongoing issues with reading sensor data, and there was no good way to know if the sensor had been switched to SPI or not. The values that could be read were either all zeroes or some static numbers. This led us to test different SPI libraries in hope of finding a suitable one. All of the SPI libraries that were tried did not work the way we wanted to, which resulted in a lot of debugging and a time-consuming coding period.

The sensor testing on the PCB was also challenging, as there was no good way of knowing if the HIROSE connector pins were properly connected to the sensors. We were in great need of a BMI085 shuttle board to be able to properly test the code, as we then would know that the connections were properly connected using jumper wires. This only became available during the second to last week of the project, but we got to test some of the code. Accessing the sensor via I2C worked without much difficult coding, and I2C is by far the easiest protocol to use. The SPI protocol was also tested on the shuttle board, but again we ran into issues. Since the pin spacing was so small, it was very difficult setting up the 9 pins needed for the SPI connection on the board. We again received all zero values from the sensor. This could be due to wrong coding and poor connection. When doing the I2C testing, it was noticed that the sensor some times would turn off when turning the sensor. Even with only 4 pin connections, that the I2C requires, the sensor had difficulties staying in contact with the Raspberry Pi. Due to this, it was also believed that the same could have occurred with the SPI setup.

It was also proven that the Wiring Pi library works with SPI, as it successfully sent IMU values from the Raspberry Pi to Arduino, and the blinker test using this library also showed the CS pin going high and low on the oscilloscope. Since the I2C protocol gave successful outputs of data, we believe this could be a viable temporary solution. An option could be to connect both the I2C and the SPI pins from the sensor to the Raspberry Pi, leaving the option of choosing either one, if the SPI code successfully can read data. The I2C protocol was also used for the BMI088 sensor in a previous iteration of the drone project [72], which also was implemented in ROS. This worked as expected, and even though SPI is faster than I2C, it would give an opportunity to test the drone without the SPI functioning. As mentioned, the HIROSE connectors play a big role in this scenario, which is also discussed in 6.2.2.

6.8 SPI libraries for the Raspberry Pi

As already discussed, the Wiring Pi library worked in some scenarios. Though it worked, it was not successful in reading the IMU values over SPI. It was noticed that when we received values, though they were static, they seemed to depend on what was provided in the buffer to the SPI transaction function. One case scenario could be that the values are simply read wrong in the code, but this seems a bit odd when the same setup was used to read the incremented value that was sent from Arduino to RPi shown in figure 4.64. There might be some issues regarding the use of a CM4 in contrast to the RPi 4 MOD B, but in essence, these share the same processor and in this application also the same OS. Since the Wiring Pi library worked on the MOD B and also worked on the CM4 when doing the test in figure 4.64, there should be no issue. Then again, it is useful to properly investigate this.

The other libraries that were tried were not tested with other sensors, so we can not say for certain that these work. The pigpio library had good documentation of some of the functions, but not all of them. In addition, there were no examples of how to use the regular SPI read and write functions, only the bit-bang portion was documented. Bit-banging is, as mentioned in section 4.11.2 not ideal for the drone application, and should the pigpio library be used, the standard SPI functions should be the main focus. As for spidev, it was expected that this would work. This was due to the fact that spidev is the default interface on the Pi. It however did not work, and here again the issue could be the HIROSE connectors or user error. This was hard to debug, and the best solution to investigate this would be to use the shuttle board with compatible jumper wires, as it was shown in section 4.11.4 that both soldering and regular jumper wires were unsuccessful. Then if a successful program is made, this could then be tested on the PCB. If it does not work here, the error is most likely located on the HIROSE connectors.

6.9 Microcontroller operations

The data transferring over SPI for the microcontroller were successful using Wiring Pi. However, we were only able to receive one value at a time, and as mentioned in section 4.13 this could be due to timing issues or that the code on the microcontroller is not sufficient enough to read all the values. The transfer is nonetheless successful, and in an ideal scenario, the microcontroller should only have to receive the computed errors from the PID to put in the motor mixer.

6.10 Debugging

After an extensive amount of testing of the sensors, tracks, and other components it is clear that there should be implemented a better solution to make testing easier. Tracks carrying for example SPI signals that go directly into the pads underneath a sensor have no contact points that can be used to debug. Multimeters and oscilloscopes are great tools to check if and what type of signal is emitted through a track. During testing of the PCB v3 it was necessary to scratch away the layer above a copper track to access it, then solder a small tap so that contact with the track could be achieved with testing equipment. A solution to this would have been to implement so-called jumper solder points along tracks that possibly would need debugging. This would have made the job to mount the components only slightly more comprehensive, but signal testing would be much easier, as they work as a contact point to the wanted tracks.

6.10.1 Bricked RPi CM4

The RPi CM4 broke towards the end of the project. Although what was already done with the software on it was very well documented, it was still a setback to the project as the RPi is supposed to be mounted on the stack inside the drone. The content and modifications made to the RPi OS and software were not backed up anywhere, so this led to an increase in work as a new one would need to be set up. Since the process of the setup is well documented it would not have resulted in a large number of problems, it just takes time. But due to the worldwide component shortage, as mentioned in 6.1, it was no way to get hold of a new RPi CM4.

Chapter 7

Conclusions

The research topic in this thesis was to design a new single board flight controller and implement it into the drone from the previous project. The thesis focuses on designing the flight controller based on a criterion of its weight, size, and performance. The flight controller should quickly react to changes in sensor values to be able to operate the drone with precision. Therefore it is evaluated which framework, software, and operating system to use when programming the flight controller. Structural changes are also made to ensure that all the drone components fit properly within the drone shell.

The flight controller is a microprocessor, namely the Raspberry Pi Compute Module 4, in combination with a diversity of external sensors mounted on a self-developed PCB. The drone body is made out of PMMA to be able to handle collisions without breaking. The mounting brackets are designed to fit within the drone shell and to keep the components firmly in place. This is to minimize vibrations within the frame, and also to protect the components in case of a crash. The PCB is designed utilizing the SPI protocol across all the main components, except for the ToF sensor which is designed to use the I2C-protocol. This is done to ensure that the quickest available interfaces for the sensor communication are implemented. The microcontroller is used for PID control and the motor mixing algorithm. The framework F' is not implemented, as the design phase proved the framework insufficient for the application. As of now, the drone stands without a framework.

Circuits boards and mechanical drone parts designed for the project were completed and ready to use, with the exception of the bricked RPi CM4. Further testing could have been continued immediately with a new RPi CM4, but it was not possible to get hold of a new one due to the shortage of electronic components.

The HIROSE connector also proved to be problematic due to the difficulty of debugging it. The small size and delicacy of the mounting led to multiple problems with placement and short-circuiting between the pins. This could have interfered with the sensor communication due to overlapping pins that were not discovered.

The SPI protocol proved itself difficult to implement. Several libraries were used without much luck. Finally, the I2C protocol worked with the shuttle board, which shows that by using the shuttle board the code can be better tested than doing it directly on the PCB. The SPI protocol is however the superior choice considering speed and efficiency, so even though I2C worked and can be used without much problems, the SPI protocol is still preferred.

The PID control and motor mixing algorithm are functioning on the microcontroller, but as mentioned, the PID control should be on the microprocessor side. The PID control is also only functioning on negative pitch and roll values, which is not a complete PID control and needs to be fixed. It also remains to tune this controller, as the drone has yet to be flown.

Some of the research done for this year's project was to investigate the possibility of using NASA's F' framework in a self-developed flight controller. The results revealed some issues with the usage of F' considering the implementation of components. The available information and tutorials about using the framework were not that good compared to what ROS has to offer. It was also not stated anywhere that the framework has been used in drone applications, whereas ROS is broadly used in this field.

The project's overall results were a little below what was expected, as the SPI interface problems led to a long period of debugging without sufficient results. Because of this, the overall performance of the flight controller could not be tested as a whole, and the drone was also not able to fly. If the SPI communication problems were resolved it should be relatively straight forward to adopt the SPI code to work with the other components that rely on it. This makes SPI along with the HIROSE connector, are two of the major holdbacks to the drone taking flight.

Due to the results of the F' testing and evaluating it alongside ROS it was concluded that ROS is the superior choice for this drone platform, and should be the main framework. The PID control should be implemented on the microprocessor side, and leave the microcontroller to only do the motor mixing. This is to utilize the RTOS on the microprocessor and decrease the load on the microcontroller.

In conclusion, a lot of valuable research for the drone project was done, leaving the next iteration of the project with good information and data to base further development on. Despite not being able to fly yet, the drone shell was completed, and the new components were mounted to our satisfaction. The project is therefore concluded with research and evaluations of which components and software it is recommended to use in further development.

Chapter 8

Further Development

During the project, a lot of work and research has been done involving drones, flight controllers, and programs. Some problems did occur and some ideas got put to life, but some did not get the chance to be realized. This chapter is the recommended improvements and changes if the project is to be further developed.

8.1 Change of the sensor interface

For further improvements on the project or on a continuation of the project, it might be useful to change the sensor interface from SPI to I2C as explained in section 6.7. Although the I2C is slower than SPI, it is easier to run and test, and might therefore be better during the development of the flight controller. As discussed in section 6.7 it could be a good solution to implement both, so that they are both available for testing. No changes would be needed on the PCB when a final solution has been reached since it has both interfaces.

8.2 Implement the Time of Flight sensor

Further testing on why the sensor reads zero in some readings will be helpful if implementing the ToF sensor to any further extent. If following that path, it is recommended to develop a self-made PCB and mount the VL53L5CX directly to that. That will reduce both cost and size. By self-developing a PCB for the ToF sensor, it is possible to use a connector type that is best suited, to make better room for the battery.

By reducing the size of the ToF sensor board, the drone is able to carry enough sensors to provide a 360-degree vision of its surroundings. A total of eight sensors are needed to support this. By having a surrounding vision the drone is able to detect any object it may collide with, and it is a matter of software development to make an anti-collision system. Doing this would make a more self-reliant autopilot.

8.3 Develop more debug possibilities

It should be considered if it would be beneficial to add solder jumper points or another form of pads that let the user tap into the tracks that send signals like SPI or I2C from one component to another. This is because it is hard to debug small embedded tracks on a carrier board as the signal travels inside the board and into the components, and there is no place to check the signal with a multimeter or an oscilloscope.

If decided to keep the HIROSE connector, it should be evaluated if a change of pad size for the connector in the footprint can be done. This might make debugging of the HIROSE easier, as individual pins could be measured and debugged. It would also make errors during the reflow process more visible, and therefore easier to find and fix.

8.4 Raspberry Pi ZERO

Considering the problems regarding the HIROSE connector it would be beneficial to the project to try to resolve these by evaluating a change from the RPi CM4 to an RPi Zero 2 W, and therefore getting rid of the HIROSE connector. Research still has to be done to ensure that the RPi Zero 2 W has a sufficient amount of SPI and I2C availability as the RPi CM4 through its GPIO pins or otherwise. The same HIROSE problems might also be solved by ordering the PCB with the HIROSE already mounted, but then it has to be controlled if the HIROSE connector can withstand two times in the reflow oven.

8.5 Further software development

Further work on the software side would be to further develop the programs for the IMU and barometer. The barometer has not been tested, and no code has been developed for it as it did not mount properly on the PCB. If a code for the IMU over SPI is successfully made, there should be no issue using this code as a baseline for developing a similar code for the barometer. It should also be looked at how these codes should be implemented on the platform. F' has proven to be a complicated framework, and one way to go is to continue to use ROS and implement the codes similar to what was done in this previous bachelors [62][72][79].

Some form of maneuvering of the drone must also be implemented. Whether this is autonomous or not could be for the next developers to decide, but the possibilities of implementing RC-controllers or mission mapping is possible to do in ROS. An RC-control would be a great asset to test the drone before a possible swarm is made. Mission mapping/Mission planning would be more towards operating and coordinating the swarm.

The microcontroller should also not have to do the PID control. This should be implemented on the Raspberry Pi, as it is much faster and also has the RTOS. It is possible to have an RTOS on the microcontroller as well, so this could be looked in to. A fitting RTOS for the microcontroller could be FreeRTOS [50]. ROS also has PID packages which could be of great use. The motor mixing algorithm that is made is fairly easy to understand and can be used further. It does not however contain yaw inputs, as there is no magnetometer on the drone. Yaw can be somewhat calculated using the implemented sensors, but the best way is to use a magnetometer.

Acronyms and abbreviations

MTOM - Maximum Takeoff Mass
VLOS - Visual line of sight
EVLOS - Extended visual line of sight
BLOS - Beyond line of sight
PCB - Printed Circuit Board
DRC - Design Rule Check
ToF - Time of Flight sensor
LDO - Low Drop-Out Regulator
RPi - Raspberry Pi
CM4 - Compute Module 4
FoV - Field of View
IMU - Inertial Measurement Unit
SOT - Small Outline Transistor
F' - F'Prime framework
DOF - Degrees of freedom
DC - Direct current
PID - Proportional Integral Derivative controller
DRC - Design Rule Check
I2C - Inter-Integrated Circuit
SPI - Serial Peripheral Interface
SCL - Serial clock
SDA - Serial Data
SS = CE = CS - Chip select
PWM - Pulse Width Modulation
GPIO - General-purpose input/output
PLA - Polyactic acid
PP - Polypropylene
SSH - Secure Shell
CLK - Clock
API - Application User Interface
VM - Virtual Machine
PMMA - Polymetylmetakrylat
MISO - Master Input Slave Output
MOSI - Master Output Slave Input
SCLK - Serial Clock
NAT - Network Address Translation
TCP - Transmission Control Protocol
GPOS - General Purpose Operating System
RTOS - Real-Time Operating System
UART - Universal asynchronous receiver-transmitter
ESC - Electronic speed control
PDU - Power distribution unit
CPU - Central Processing Unit
POSIX - Portable Operating System Interface

SYSV - Unix System 5
GUI - Ground User Interface
COINES - COmmunication with INertial and Environmental Sensors
IOCTL - Input Output Control
PS - Protocol Selection
LSB - Least Significant Byte
MSB - Most Significant Byte
IDE - Integrated Development Environment

Appendix A

Patching the Linux kernel to real-time

The kernel in the original Raspberry OS is not a real-time kernel. This means that to envoke the real-time capabilities, one has to patch the kernel with the PREEMPT_RT patch from Linux. This is a step by step guide to do this. The guide is done using Ubuntu 20.04 LTS as a host system, with the Raspberry Pi 4 Model B as target. Before starting, make sure to have installed the cross-compilation tools for Raspberry Pi as well as menuconfig. Also notice that this is a general guide, and there was experienced problems with lacking of files from the original RPi Linux repository. This is discussed further in A.3.

A.1 Building the kernel

This guide is based off of the guide provided by Raspberry Pi [99]. First off, one needs to know the kernel version of the target system, in this case the Raspberry Pi. To find this, execute the command uname -r in the target terminal. This will give you three numbers.

```
pi@raspberry4:~ $ uname -r
5.10.92-v8+
```

As one can see, the kernel version is version 5 with patch number 10 and sublevel 92. One then has to make sure that the real time patch being built is matching this kernel version.

To execute the build in a clean way, make a designated directory for the kernel. In this case, it is named rpi-kernel and put in the home directory of the host machine.

```
host@hostname:~$ mkdir rpi-kernel
```

The patching can also be done directly on the Raspberry, but for a more efficient build we use the host machine as it is way more powerful than the Pi. When the patching is done one can either cross compile the images to the Pi or put them directly in the boot folder of the SD-card.

Now, cd into the rpi-kernel folder or the name you gave the directory. Use the git clone command to clone the linux repository for raspberry pi into the directory. Then activate the branch of the current kernelversion on the raspberry found in the previous step. To do this, use the git checkout command.

```
host@hostmachine~/rpi-kernel$ git clone ...
   https://github.com/raspberrypi/linux.git
host@hostmachine~/rpi-kernel$ cd linux
host@hostmachine~/rpi-kernel/linux$ git checkout rpi-5.10.y
```

A.2 Applying the PREEMPT_RT patch

It is important to note that the default kernel configuration when cloning the 5.10.y repo is 5.10.95. Since we want the 5.10.90 kernel branch, as this is the closest branch with a real time patch, one can clone this specific repo by locating it in the kernel.org linux tree and using wget to download it. This to ensure that no error occurs due to mismatching configuration and the rt-patch. The kernel branch used in this thesis is located here [66]. In addition to this, as mentioned, the Raspberry pi has kernel version 5.10.92. To make sure that no error occur here either, one can downgrade the kernel version. This is done using the rpi-update command directly on the raspberry pi. Combine this command with the git hash of the commit for kernel version 5.10.90 to get the correct kernel version on the Pi. The commits can be found in the firmware section of the raspberry pi repository on Github [107].

```
host@hostmachine:~/rpi-kernel$ wget ...
    https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/snapshot/linux-d3e
     491a20d152e5fba6c02a38916d63f982d98a5.tar.gz
host@hostmachine:~/rpi-kernel$ gunzip ...
    linux-d3e491a20d152e5fba6c02a38916d63f982d98a5.tar.gz
```

Now the repository is downloaded and the files are configured to match the kernelversion on the Pi. The next step is to find the realtime patch for the kernel. Go to the kernel.org/linux website and locate the link-address of the matching realtime patch [65]. In this case, the closest available patch is 5.10.90. While in the linux directory, download this patch using the wget command. Then unzip the patch by executing the gunzip-wizard. To apply the patch, we use the cat command.

```
host@hostmachine:~/rpi-kernel/linux$ wget ...
    https://mirrors.edge.kernel.org/pub/linux/kernel/projects/rt/5.10/patch-5.10.90-
     rt61.patch.gz
host@hostmachine:~/rpi-kernel/linux$ gunzip patch-5.10.90-rt61.patch.gz
host@hostmachine:~/rpi-kernel/linux$ cat patch-5.10.90-rt61.patch.gz | ...
    patch -p1
```

```
pi@raspberrypi:~ sudo rpi-update 9a09c1dcd4fae55422085ab6a87cc650e68c4181
```

Once the patch is applied, one has to activate the Preemption model. This is done using menuconfig. Since the RPI 4 Mod B and the RPI CM4 are x64 systems, use the x64 cross-compiler and kernel8. To choose kernel8, write export KERNEL=kernel8 in the terminal window. Since we are cross-compiling a 64-bit kernel, the make command must specify this when we make the menuconfig. If not it will just activate the default configuration bcm2711_defconfig. One would then not be able to choose the correct preemtion model right away. To avoid this, we specify that we want to cross-compile to a 64-bit system by executing the following command in the linux directory.

```
host@hostmachine~/rpi-kernel/linux$ make ARCH=arm64 ...
    CROSS_COMPILE=aarch64-linux-gnu- menuconfig
```

This generates a .config file, which will be used to build the kernel. When the above command is executed, it will open the menuconfig window in the terminal. To the activate the correct Preemption model, navigate to General Setup → Preemption Model and choose the Fully Preemptible Kernel (Real-Time) option as shown in figure A.1. Save the changes and exit the configuration.

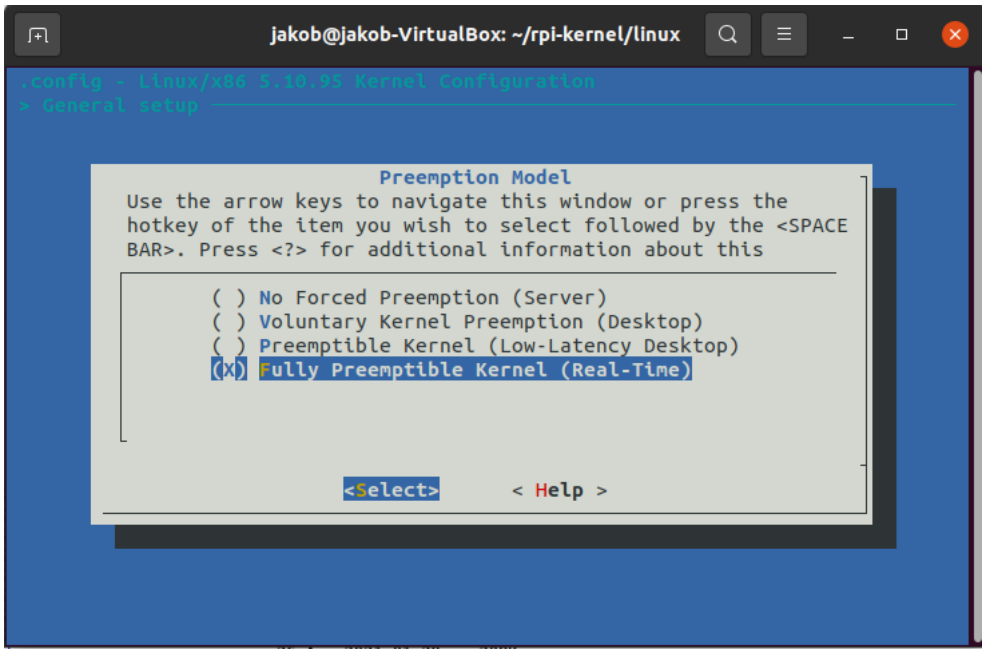


Figure A.1: Preemption Model in menuconfig

A.3 Building with the configurations

Now that the configurations are applied, we can build the kernel. All the patch changes are applied in the .config file which is executed during the build. To start the build process, execute the following line in the terminal window. Since this is a virtual machine, one can give the VM more computer power than just one processor. In this case we give it 4 processors, and execute -j4 together with the make command to speed up the building process.

```
host@hostmachine~/rpi-kernel/linux$make ARCH=arm64 ...
    CROSS_COMPILE=aarch64-linux-gnu- -j4 Image modules dtbs
```

If any errors occur, try to build the different aspects by themselves. Start with the Image, then modules and then dtbs. This can give more info about the error and how to fix it.

In the particular build case conducted in this thesis, it was experienced that the 5.10.y repository from github was insufficient. When building the datablods (.dtb) it seemed like the command was carried out as no error message popped up, but after trying to deploy the kernel it was noticed that these files were missing and therefore resulting in a black screen when rebooting the Pi with the fresh kernel. To build the kernel properly, a fork of the 5.10.y repository was used [64]. This repository contained a pre-configured 64-bit deployment of the patch, and was much easier to build. The guide provided above is a general way of patching the kernel, and similar steps can be followed to build the forked kernel deployment. The .config file generated was entirely similar to the one from kdoren with the exception that the makefile from kdoren actually included the device tree path from the RPi repository, while the directly cloned repository didnt.

```
host@hostmachine~/rpi-kernel$ git clone ...
    https://github.com/kdoren/linux.git -b
rpi-5.10.90-rt
```

When the repo is cloned, cd into the repo and find the file named rt64.conf. Rename this to .conf and execute the same commands as previously mentioned to make the menuconfig. When this is done, one can see that the patch is already applied to the config file and Fully Preemptible kernel is chosen. Now choose the same arcitechture and cross compiler as before and make the Image,

modules and dtbs. While this is running, create an empty folder outside the linux folder. This will be used for cross compiling the kernel to the Pi. Cd back into the linux folder and execute the following commands:

```
host@hostmachine~/rpi-kernel/linux export ...
    INSTALL_MOD_PATH=PATH_TO_EMPTY_FOLDER
host@hostmachine~/rpi-kernel/linux export ...
    INSTALL_DTBS_PATH=PATH_TO_EMPTY_FOLDER
```

The name you gave the empty folder you made should be at the other end of the equals sign. This tells the compiler to compile the modules and datablobs needed to boot the kernel in to the empty folder you made. When all the builds are done, execute the same commands but now building the modules_install and dtbs_install. This builds the necessary module and datablob-files to the empty folder you made earlier.

```
host@hostmachine~/rpi-kernel/linux$make ARCH=arm64 ...
    CROSS_COMPILE=aarch64-linux-gnu- -j4 modules_install dtbs_install
```

This should give 3 folders in the empty folder you made. Broadcom, lib and overlays. Now, make another folder inside this empty folder named boot. Now, you will copy the kernel image built earlier, by executing this command:

```
host@hostmachine~/rpi-kernel/linux cp /arch/arm64/boot/Image ...
    $INSTALL_MOD_PATH/boot/$KERNEL_rt.img
```

You should now have a folder with all the necessary files to deploy the kernel. Make a .tgz zipped file of the folder and copy it to your Pi by using the scp command. Copy this to the /tmp folder on the pi, and unzip the contents.

```
host@hostmachine~/rpi-kernel/ scp name_of_folder.tgz pi@Pi_ip_address:/tmp
pi@raspberrypi~/tmp$ tar xzf rt-kernel.tgz
```

Now execute the following commands to copy all the files into the correct folders.

```
/tmp$ cd boot
/tmp/boot$ sudo cp -rd * /boot/
/tmp/boot$ cd ../lib
/tmp/lib$ sudo cp -dr * /lib/
/tmp/lib$ cd ../overlays
/tmp/overlays$ sudo cp -d * /boot/overlays
/tmp/overlays$ cd ..
/tmp$ sudo cp -d bcm* /boot/
```

When this is finished, reboot the Pi and type uname -r in the terminal to check if the patch has been applied. In this case it is displayed that the 5.10.90-rt61 kernel is enables, but this depends entirely on what patch was applied. Thats it! The kernel is now operating in real time. 18

A.3.1 Common Errors

If you run the build processes seperately, and an error pops up, the terminal usually tells you what needs to be changed. The errors that occured during this build were the following:

- CONFIG SYSTEM TRUSTED KEYS
  Solution: Set to " " by removing debian/canonical-certs.pem
- CONFIG SYSTEM REVOCATION KEYS
  Solution: Set to " " by removing debian/canonical-revoked-certs.pem

Appendix B

Matlab Code

B.1 Ptsematest

Listing B.1: Ptsematest

```
%% ptsematest
clear; close all; clc;

T = readtable('datalog_regular_ptsematest.txt');

    max_0 = T(5:8:end,:);
    max_1 = T(6:8:end,:);
    max_2 = T(7:8:end,:);
    max_3 = T(8:8:end,:);

    figure
    plot(max_0.Var11, LineWidth=2)
    xlabel('Iterations')
    ylabel('Time in us')

    hold on
    plot(max_1.Var11, LineWidth=2)

    hold on
    plot(max_2.Var11, LineWidth=1.5)

    hold on
    plot(max_3.Var11, LineWidth=1.5)

    legend('CPU 0', 'CPU 1', 'CPU 2', 'CPU 3')
    title('ptsematest regular kernel 5.10.90')
```

```
%% ptsematest realtime
clear; close all; clc;

T = readtable('datalog_rt_ptsematest.txt');

    max_0 = T(5:8:end,:);
    max_1 = T(6:8:end,:);
    max_2 = T(7:8:end,:);
    max_3 = T(8:8:end,:);

    figure
    plot(max_0.Var11, LineWidth=2)
    xlabel('Iterations')
    ylabel('Time in us')
```

```
    hold on
    plot(max_1.Var11, LineWidth=2)

    hold on
    plot(max_2.Var11, LineWidth=1.5)

    hold on
    plot(max_3.Var11, LineWidth=1.5)

    legend('CPU 0', 'CPU 1', 'CPU 2', 'CPU 3')
    title('ptsematest RT-kernel 5.10.90-rt61')
```

B.2 Sigwaittest

Listing B.2: Sigwaittest

```
%% sigwaittest
clear; close all; clc;

T = readtable('datalog_regular_sigwaittest.txt');

    max_0 = T(5:8:end,:);
    max_1 = T(6:8:end,:);
    max_2 = T(7:8:end,:);
    max_3 = T(8:8:end,:);

    figure
    plot(max_0.Var11, LineWidth=2)
    xlabel('Iterations')
    ylabel('Time in us')

    hold on
    plot(max_1.Var11, LineWidth=2)

    hold on
    plot(max_2.Var11, LineWidth=1.5)

    hold on
    plot(max_3.Var11, LineWidth=1.5)

    legend('CPU 0', 'CPU 1', 'CPU 2', 'CPU 3')
    title('sigwaittest regular kernel 5.10.90')
```

```
%% sigwaittest realtime
clear; close all; clc;

T = readtable('datalog_rt_sigwaittest.txt');


    max_0 = T(5:8:end,:);
    max_1 = T(6:8:end,:);
    max_2 = T(7:8:end,:);
    max_3 = T(8:8:end,:);

    figure
    plot(max_0.Var11, LineWidth=2)
    xlabel('Iterations')
```

```
    ylabel('Time in us')

    hold on
    plot(max_1.Var11, LineWidth=2)

    hold on
    plot(max_2.Var11, LineWidth=1.5)

    hold on
    plot(max_3.Var11, LineWidth=1.5)

    legend('CPU 0', 'CPU 1', 'CPU 2', 'CPU 3')
    title('sigwaittest rt-kernel 5.10.90-rt61')
```

B.3 Svsematest

Listing B.3: Svsematest

```
%% svsematest
clear; close all; clc;
T = readtable('datalog_regular_svsematest.txt');

    max_0 = T(5:8:end,:);
    max_1 = T(6:8:end,:);
    max_2 = T(7:8:end,:);
    max_3 = T(8:8:end,:);

    figure
    plot(max_0.Var11, LineWidth=2)
    xlabel('Iterations')
    ylabel('Time in us')

    hold on
    plot(max_1.Var11, LineWidth=2)

    hold on
    plot(max_2.Var11, LineWidth=1.5)

    hold on
    plot(max_3.Var11, LineWidth=1.5)

    legend('CPU 0', 'CPU 1', 'CPU 2', 'CPU 3')
    title('svsematest regular kernel 5.10.90')
```

```
%% svsematest rt
clear; close all; clc;
T = readtable('datalog_rt_svsematest.txt');

    max_0 = T(5:8:end,:);
    max_1 = T(6:8:end,:);
    max_2 = T(7:8:end,:);
    max_3 = T(8:8:end,:);

    figure
    plot(max_0.Var11, LineWidth=2)
    xlabel('Iterations')
    ylabel('Time in us')
```

```
    hold on
    plot(max_1.Var11, LineWidth=2)

    hold on
    plot(max_2.Var11, LineWidth=1.5)

    hold on
    plot(max_3.Var11, LineWidth=1.5)

    legend('CPU 0', 'CPU 1', 'CPU 2', 'CPU 3')
    title('svsematest rt-kernel 5.10.90-rt61')
```

B.4 N-Queens problem

Listing B.4: N-Queens problem

```
%% N-Queens problem regular kernel
clear; close all; clc;
T = readtable('multithread_output_regular');

    temp = T.cpu_temp;
    time_all_solutions = 0:0.26111:2.35;
    time_per_iteration = T.seconds;

    yyaxis left
    scatter(time_all_solutions, time_per_iteration, 'blue', 'filled')
    ylabel('Time to find solutions in seconds')

    hold on

    yyaxis right
    ylabel('CPU temperature')
    plot(time_all_solutions,temp, 'red', LineWidth=2)

    legend('Solution time', 'CPU Temperature')
    xlabel('Total timeduration in minutes')
```

```
%% N-queens problem rt-kernel
clear; close all; clc;
T = readtable('multithread_output_rt');

    temp = T.cpu_temp;
    time_all_solutions = 0:0.2648111:2.3833;
    time_per_iteration = T.seconds;

    yyaxis left
    scatter(time_all_solutions, time_per_iteration, 'blue', 'filled')
    ylabel('Time to find solutions in seconds')

    hold on

    yyaxis right
    ylabel('CPU temperature')
    plot(time_all_solutions,temp, 'red', LineWidth=2)

    legend('Solution time', 'CPU Temperature')
    xlabel('Total timeduration in minutes')
    title('N-Queens problem RT-kernel')
```

B.5 TOF Sensor readings

Listing B.5: ToF Sensor

```
%% TOF-sensor readings
clear; close all; clc;
T = readtable('ToF.xltx');

reading = 0 : 1 : 15;

acryl1 = T.Var2;
acryl2 = T.Var3;
acryl3 = T.Var4;
acryl4 = T.Var5;
acryl5 = T.Var6;
acryl6 = T.Var7;
acryl7 = T.Var8;
acryl8 = T.Var9;
acryl9 = T.Var10;


figure

plot(reading, acryl1, '.', 'MarkerSize', 10, 'Color', 'b');
hold on
plot(reading, acryl2, '.', 'MarkerSize', 10, 'Color', 'b');
hold on
plot(reading, acryl3, '.', 'MarkerSize', 10, 'Color', 'b');
hold on
plot(reading, acryl4, '.', 'MarkerSize', 10, 'Color', 'b');
hold on
plot(reading, acryl5, '.', 'MarkerSize', 10, 'Color', 'b');
hold on
plot(reading, acryl6, '.', 'MarkerSize', 10, 'Color', 'b');
hold on
plot(reading, acryl7, '.', 'MarkerSize', 10, 'Color', 'b');
hold on
plot(reading, acryl8, '.', 'MarkerSize', 10, 'Color', 'b');
hold on
xlabel('Reading 0 to 15');
ylabel('Distance in mm');
title('Measurements with Acrylic glass');
plot(reading, acryl9, '.', 'MarkerSize', 10, 'Color', 'b');

%% Without acryl
clear; close all; clc;
T = readtable('ToF2.xlsx');

reading = 0 : 1 : 15;

acryl1 = T.Var12;
acryl2 = T.Var13;
acryl3 = T.Var14;
acryl4 = T.Var15;
acryl5 = T.Var16;
acryl6 = T.Var17;
acryl7 = T.Var18;
acryl8 = T.Var19;
acryl9 = T.Var20;
acryl10 = T.Var21;
```

```
figure

plot(reading, acryl1, '.', 'MarkerSize', 10, 'Color', 'b');
hold on
plot(reading, acryl2, '.', 'MarkerSize', 10, 'Color', 'b');
hold on
plot(reading, acryl3, '.', 'MarkerSize', 10, 'Color', 'b');
hold on
plot(reading, acryl4, '.', 'MarkerSize', 10, 'Color', 'b');
hold on
plot(reading, acryl5, '.', 'MarkerSize', 10, 'Color', 'b');
hold on
plot(reading, acryl6, '.', 'MarkerSize', 10, 'Color', 'b');
hold on
plot(reading, acryl7, '.', 'MarkerSize', 10, 'Color', 'b');
hold on
plot(reading, acryl8, '.', 'MarkerSize', 10, 'Color', 'b');
hold on
plot(reading, acryl10, 'Color', 'black');
xlabel('Reading 0 to 15');
ylabel('Distance in mm');
title('Measurements without Acrylic glass');
plot(reading, acryl9, '.', 'MarkerSize', 10, 'Color', 'b');
```

Appendix C

Ground Station Pictures

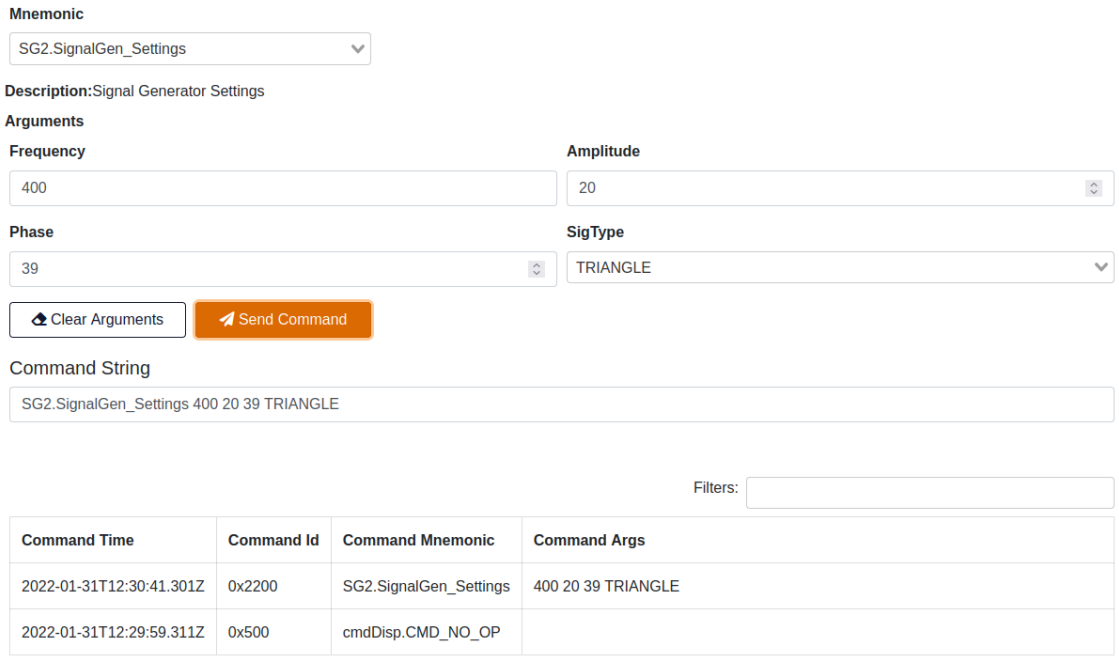


Figure C.1: Command window

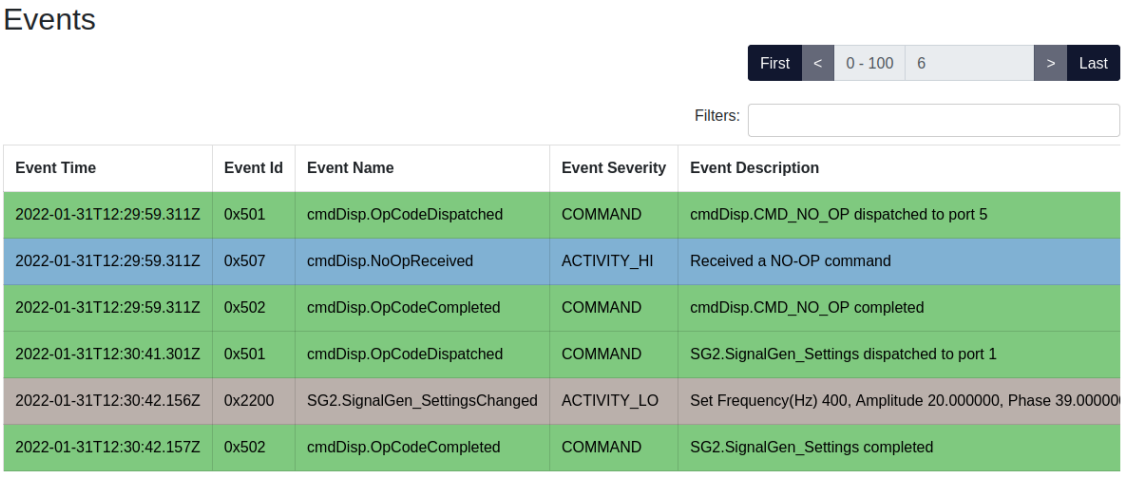


Figure C.2: Event window

Channels

Filters:

Last Sample Time	Channel Id	Channel Name	Channel Value
2022-01-31T12:31:35.186Z	0x100	blockDrv.BD_Cycles	228
2022-01-31T12:29:59.132Z	0x300	rateGroup2Comp.RgMaxTime	668 us
2022-01-31T12:29:59.132Z	0x400	rateGroup3Comp.RgMaxTime	722 us
2022-01-31T12:30:41.302Z	0x500	cmdDisp.CommandsDispatched	2
2022-01-31T12:31:31.185Z	0xa00	pingRcvr.PR_NumPings	56
2022-01-31T12:30:42.156Z	0x2200	SG2.Type	TRIANGLE
2022-01-31T12:31:35.186Z	0x2604	sendBuffComp.SendState	SEND_IDLE
2022-01-31T12:30:03.133Z	0x4402	fileUplinkBufferManager.HiBuffs	1
2022-01-31T12:31:35.186Z	0x4b00	systemResources.MEMORY_TOTAL	8143084 KB
2022-01-31T12:31:35.186Z	0x4b01	systemResources.MEMORY_USED	4984088 KB
2022-01-31T12:31:35.186Z	0x4b02	systemResources.NON_VOLATILE_TOTAL	102168536 KB
2022-01-31T12:31:35.186Z	0x4b03	systemResources.NON_VOLATILE_FREE	79343324 KB

Figure C.3: Channels window

Available Logs

channel.log

```
2022-01-31 13:27:49,(2(0)-1643632069:61139),blockDrv.BD_Cycles,256,2
2022-01-31 13:27:49,(2(0)-1643632069:61254),systemResources.CPU_00,19205,44.05 percent
2022-01-31 13:27:49,(2(0)-1643632069:61256),systemResources.CPU,19204,44.05 percent
2022-01-31 13:27:49,(2(0)-1643632069:61281),systemResources.MEMORY_TOTAL,19200,8143084 KB
2022-01-31 13:27:49,(2(0)-1643632069:61282),systemResources.MEMORY_USED,19201,4411708 KB
2022-01-31 13:27:49,(2(0)-1643632069:61287),systemResources.NON_VOLATILE_FREE,19203,79343160 KB
2022-01-31 13:27:49,(2(0)-1643632069:61287),systemResources.NON_VOLATILE_TOTAL,19202,102168536 KB
2022-01-31 13:27:49,(2(0)-1643632069:61290),systemResources.VERSION,19221,v3.0.0-RC2-197-g7e20febd3
2022-01-31 13:27:49,(2(0)-1643632069:61310),sendBuffComp.SendState,9732,SEND_IDLE
2022-01-31 13:27:50,(2(0)-1643632070:63642),systemResources.CPU_00,19205,1.00 percent
2022-01-31 13:27:50,(2(0)-1643632070:63644),systemResources.CPU,19204,1.00 percent
2022-01-31 13:27:50,(2(0)-1643632070:63679),systemResources.MEMORY_TOTAL,19200,8143084 KB
2022-01-31 13:27:50,(2(0)-1643632070:63680),systemResources.MEMORY_USED,19201,4412252 KB
2022-01-31 13:27:50,(2(0)-1643632070:63686),systemResources.NON_VOLATILE_FREE,19203,79343152 KB
2022-01-31 13:27:50,(2(0)-1643632070:63687),systemResources.NON_VOLATILE_TOTAL,19202,102168536 KB
2022-01-31 13:27:50,(2(0)-1643632070:63689),systemResources.VERSION,19221,v3.0.0-RC2-197-g7e20febd3
2022-01-31 13:27:50,(2(0)-1643632070:63692),rateGroup1Comp.RgMaxTime,512,1924 us
2022-01-31 13:27:50,(2(0)-1643632070:61791),blockDrv.BD_Cycles,256,3
2022-01-31 13:27:51,(2(0)-1643632071:62725),blockDrv.BD_Cycles,256,4
2022-01-31 13:27:51,(2(0)-1643632071:62802),systemResources.CPU_00,19205,100.00 percent
2022-01-31 13:27:51,(2(0)-1643632071:62803),systemResources.CPU,19204,100.00 percent
2022-01-31 13:27:51,(2(0)-1643632071:62819),systemResources.MEMORY_TOTAL,19200,8143084 KB
2022-01-31 13:27:51,(2(0)-1643632071:62820),systemResources.MEMORY_USED,19201,4522720 KB
2022-01-31 13:27:51,(2(0)-1643632071:62823),systemResources.NON_VOLATILE_FREE,19203,79353200 KB
2022-01-31 13:27:51,(2(0)-1643632071:62824),systemResources.NON_VOLATILE_TOTAL,19202,102168536 KB
2022-01-31 13:27:51,(2(0)-1643632071:62825),systemResources.VERSION,19221,v3.0.0-RC2-197-g7e20febd3
2022-01-31 13:27:51,(2(0)-1643632071:62838),sendBuffComp.SendState,9732,SEND_IDLE
2022-01-31 13:27:52,(2(0)-1643632072:63924),systemResources.CPU_00,19205,100.00 percent
2022-01-31 13:27:52,(2(0)-1643632072:63925),systemResources.CPU,19204,100.00 percent
2022-01-31 13:27:52,(2(0)-1643632072:63944),systemResources.MEMORY_TOTAL,19200,8143084 KB
2022-01-31 13:27:52,(2(0)-1643632072:63945),systemResources.MEMORY_USED,19201,4677876 KB
2022-01-31 13:27:52,(2(0)-1643632072:63949),systemResources.NON_VOLATILE_FREE,19203,79353200 KB
2022-01-31 13:27:52,(2(0)-1643632072:63949),systemResources.NON_VOLATILE_TOTAL,19202,102168536 KB
```

Figure C.4: Log window

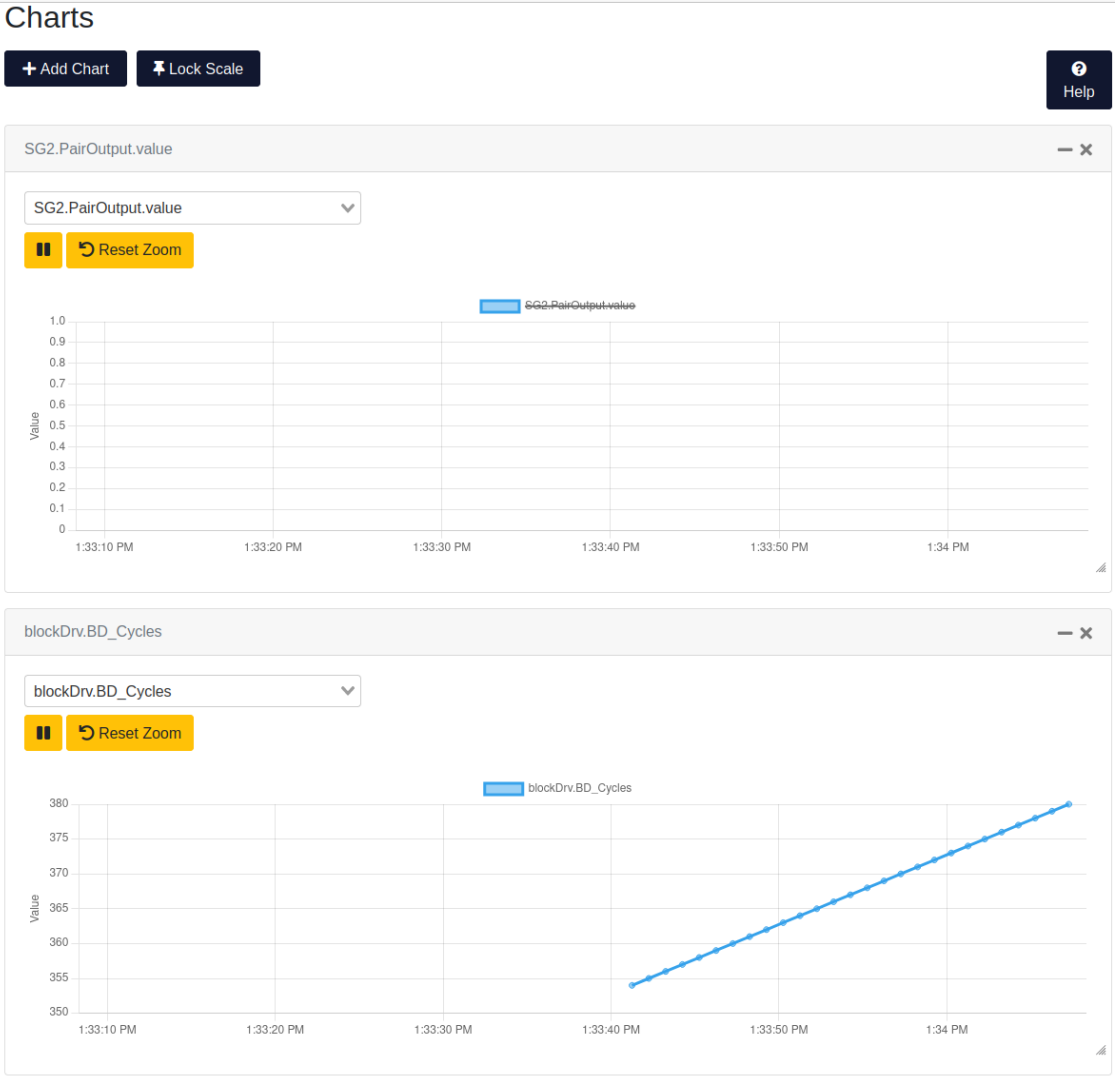


Figure C.5: Charts window

Appendix D

Port allowing and forwarding for Raspberry Pi and host machine

D.1 Port allowing in Windows Defender Firewall

Navigate to the advanced settings for the Windows Firewall. This is found by opening the control panel, then go to System and Security → Windows Defender Firewall → Advanced Settings in the leftmost row. This opens the window shown in figure D.1. Now press "New Rule", TCP and select Port and enter port 22, 80, 443, 5000 and 50000. These ports are common communication ports for the Raspberry Pi, and allowing all of them rules out any errors maybe occurring in the future. Now press next and allow the connection and give it a suitable name. In this case it is called "PI2UBUNTU" as shown in figure D.2.

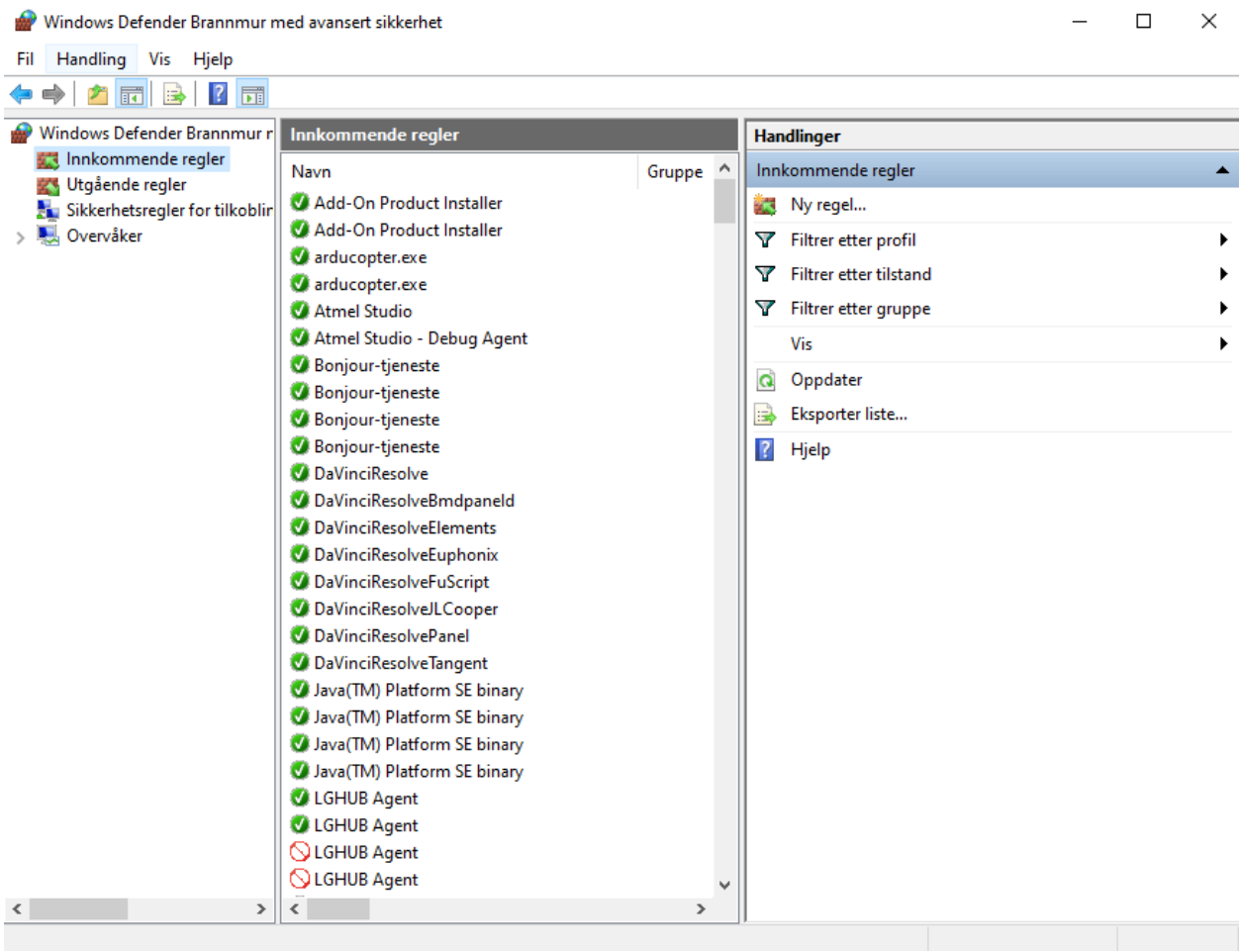


Figure D.1: Windows Defender interface

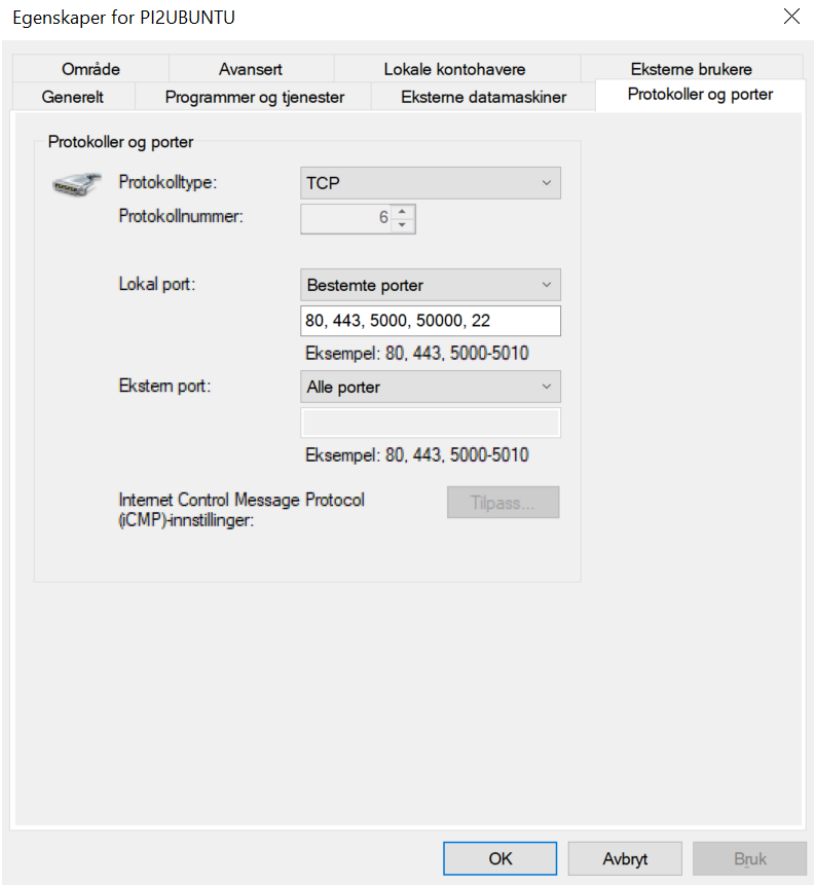


Figure D.2: Finalized port allowing for PI2UBUNTU

D.2 Port allowing and forwarding in Ubuntu and RPI

Now using the VM, use the ufw status command to check what ports are allowed on the VM. If port 22 and 50000 are not there, use the ufw allow command followed by the desired port to tell the firewall to let the connection through. This should also be done on the Pi, to ensure that the ports in use are running and that the Pi is listening to port 22 and 50000. Port 22 is as mentioned in the theory chapter, the default SSH-port. To check what ports both the Pi and the VM are listening to, run the netstat command D.3.

Listing D.1: Allowing and testing port connections

```
host@hostmachine:~ sudo ufw allow 22/tcp 50000/tcp
host@hostmachine:~ sudo ufw status
host@hostmachine:~ sudo netstat -tnl
```

By writing /tcp after the port specification, we specify that we are communicating over TCP protocol. The status command then shows all the active ports, and running the netstat command with -tnl allows one to see which TCP ports the system is listening to plus the numerical address.

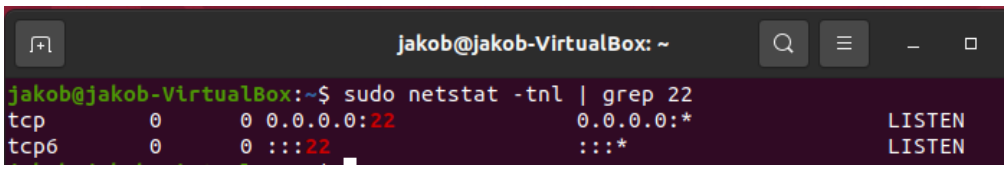


Figure D.3: Host is listening to port 22

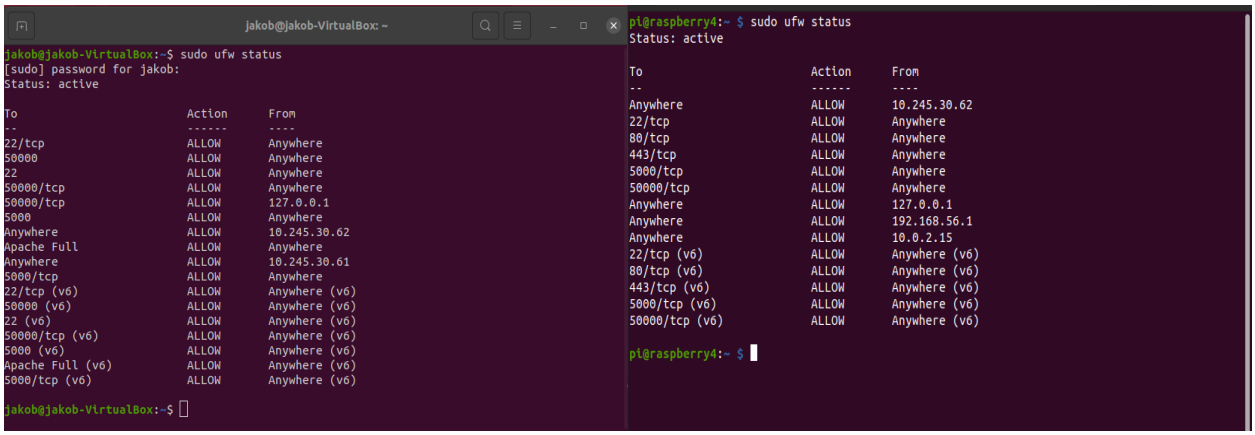


Figure D.4: Firewall status of host and RPi

In addition to this, when using Virtualbox, one has to open the port in the Virtualbox Manager as well. If choosing to use a bridged adapter connection, this is not nessecary, but during this project many problems occured trying to set up the bridged adapter. It led to no internet connection for the Virtual Machine, and thus it was decided to use the standard NAT protocol. To allow port 50000 through the Manager, access the network settings and select NAT. Then move to advanced settings, and click on "Port Forwarding" as in figure D.5. Then click the green plus sign in the right corner to make a new rule. Set this up as shown in figure D.6 where protocol is TCP and host and guest port is 50000.

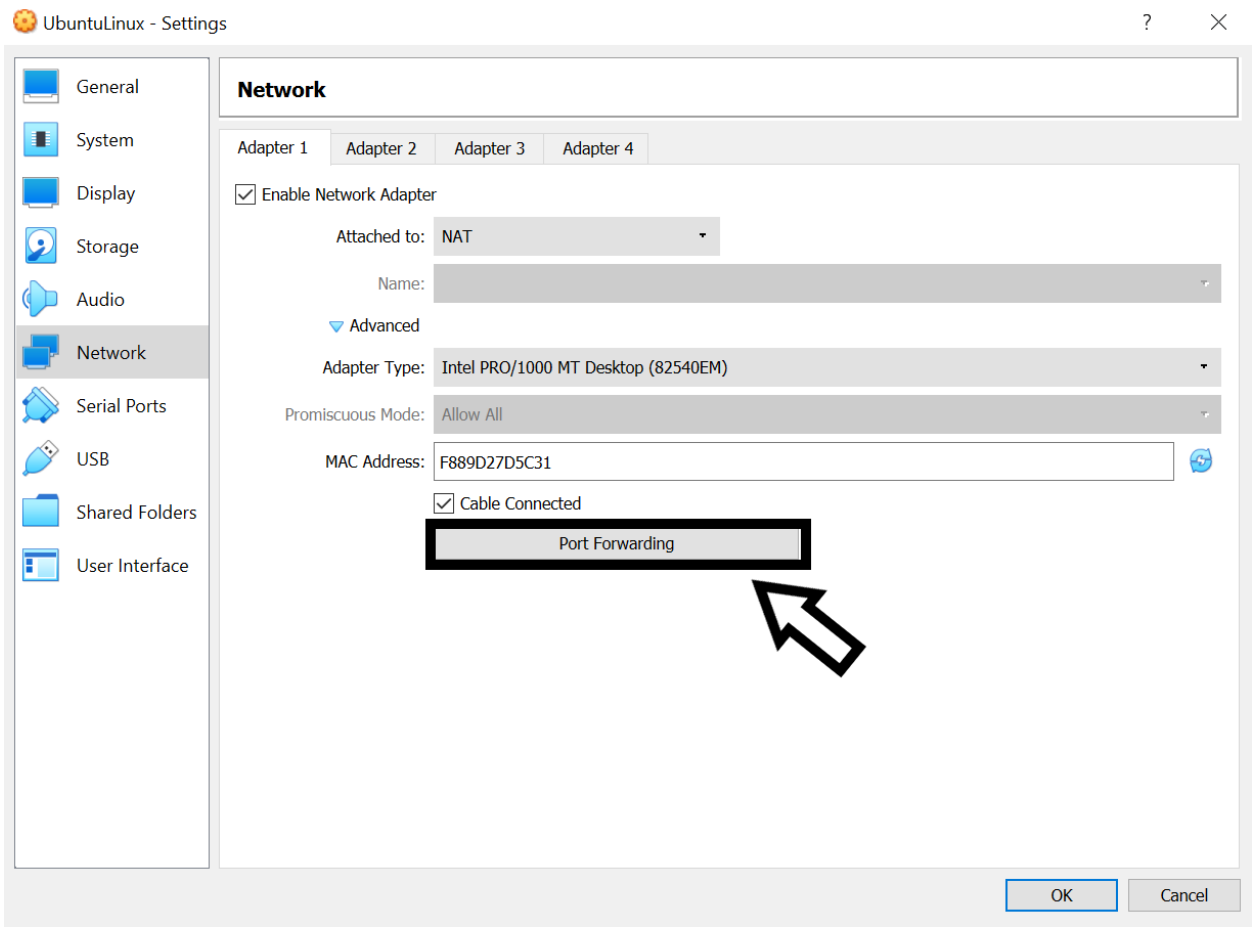


Figure D.5: Port Forwarding in VBox Manager

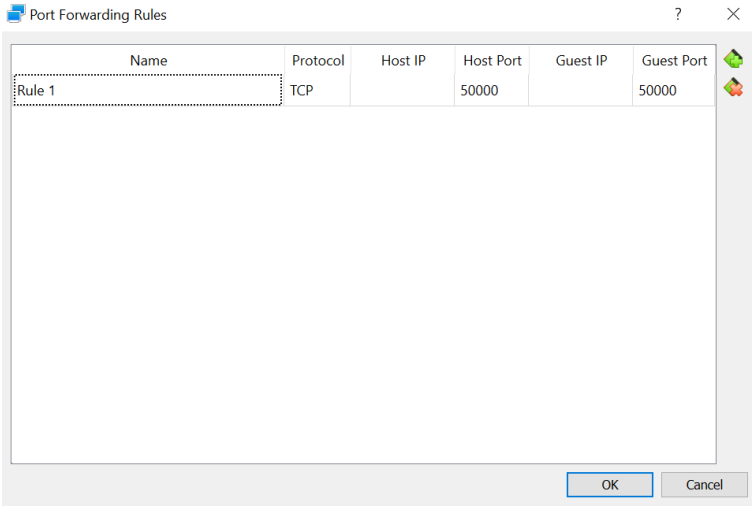


Figure D.6: New rule allowing port 50000

After this process is done, the connection to the Pi from host and vice verca can be checked by pinging the IP-addresses respectively. Using the ping command in Windows command window or the Ubuntu terminal, the connection and also response time can be tested as shown in figure D.7 bellow. Note that the ping testing in this figure was done before configuring the static IP-address for the Pi. As one can see, the pi was pinged by name, but it can also be pinged by IP-address by writing this after the ping command in substitution of the target name. Now the network configurations are done, and as one final check, SSH into the Raspberry Pi and verify the connection. This is done in figure D.8. and notice that the name of the pi now has changed to raspberry4.

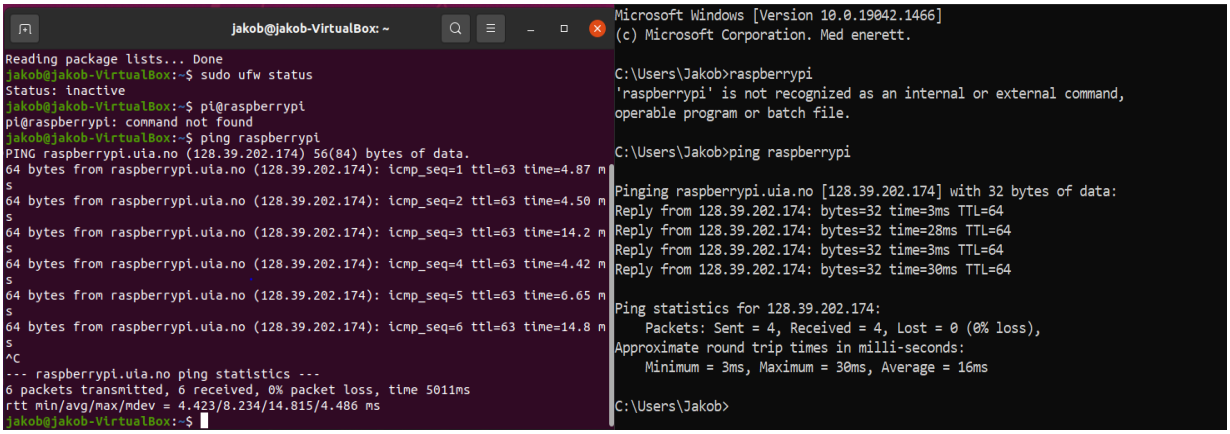


Figure D.7: Ping response from RPi

Figure D.8: SSH connection verification

Appendix E

F' Component building

E.1 Component implementation

Listing E.1: BarometerComponentAi.xml

```
<?xml version="1.0" encoding="UTF-8"?>


<component name="barometer" kind="active" namespace="BarometerApp" ...
   modeler="true">
    <!-- Import command ports -->
    <import_port_type>Fw/Cmd/CmdPortAi.xml</import_port_type>
    <import_port_type>Fw/Cmd/CmdRegPortAi.xml</import_port_type>
    <import_port_type>Fw/Cmd/CmdResponsePortAi.xml</import_port_type>
    <!-- Import event ports -->
    <import_port_type>Fw/Log/LogPortAi.xml</import_port_type>
    <import_port_type>Fw/Log/LogTextPortAi.xml</import_port_type>
    <!-- Import telemetry ports -->
    <import_port_type>Fw/Tlm/TlmPortAi.xml</import_port_type>
    <import_port_type>Fw/Buffer/BufferSendPortAi.xml</import_port_type>
    <import_port_type>Drv/I2CDriverPorts/I2CPortAi.xml</import_port_type>
    <!-- Import command, telemetry, and event dictionaries -->
    <import_dictionary>Barometer/Commands.xml</import_dictionary>
    <import_dictionary>Barometer/Telemetry.xml</import_dictionary>
    <import_dictionary>Barometer/Events.xml</import_dictionary>

    <ports>
        <!-- Command port definitions: command input receives commands, ...
           command reg out, and response out are
        ports used to register with the command dispatcher, and return ...
           responses to it -->
        <port name="cmdIn" data_type="Fw::Cmd"  kind="input" role="Cmd"    ...
           max_number="1">
        </port>
        <port name="cmdRegOut" data_type="Fw::CmdReg" kind="output" ...
           role="CmdRegistration" max_number="1">
        </port>
        <port name="cmdResponseOut" data_type="Fw::CmdResponse" ...
           kind="output" role="CmdResponse" max_number="1">
        </port>
        <!-- Event ports: send events, and text formatted events -->
        <port name="eventOut" data_type="Fw::Log"  kind="output" ...
           role="LogEvent"  max_number="1">
        </port>
        <port name="textEventOut" data_type="Fw::LogText" kind="output" ...
           role="LogTextEvent" max_number="1">
```

```
        </port>
        <!-- Telemetry ports -->
        <port name="tlmOut" data_type="Fw::Tlm"  kind="output" ...
            role="Telemetry" max_number="1">
        </port>
        <port name="I2cWrite" data_type="Drv::I2c" kind="output" ...
            max_number="1">
        </port>
        <port name="I2cRead" data_type="Drv::I2c" kind="output" ...
            max_number="1">
        </port>
        <port name="I2cBufferOut" data_type="Fw::BufferSend" kind="output" ...
            max_number="1">
        </port>
    </ports>
</component>
```

Listing E.2: Commands.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- GPS Tutorial: RPI/Barometer/Commands.xml // Changed from the tutorial ...
    script

<commands>
    <!-- Define a single command that runs asynchronously on the ...
        component's own thread. The opcode "0" is relative to
    the GPS component's command space.  The mnemonic is the string a user ...
        will use to refer to this command. -->
    <command kind="async" opcode="0" mnemonic="Barometer_status" >
        <comment>A command to force a lock status reporting.</comment>
    </command>
</commands>
```

Listing E.3: Telemetry.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- GPS Tutorial: RPI/Barometer/Telemetry.xml

This defines three telemetry channels to report basic barometer information.
-->
<telemetry>
    <channel id="0" name="Pressure" data_type="F32" abbrev="Barometer-0000">
        <comment>The current measured pressure</comment>
    </channel>
    <channel id="1" name="Celcius" data_type="F32" abbrev="Barometer-0001">
        <comment>The current temperature in C</comment>
    </channel>
    <channel id="2" name="Fahrenheit" data_type="F32" abbrev="Barometer-0002">
        <comment>The current temperature in F</comment>
    </channel>
</telemetry>
```

Listing E.4: Events.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- GPS Tutorial: RPI/Barometer/Events.xml

This defines two events, one at activity hi level to report that lock has ...
    been acquired, and one at warning hi level to
indicate lock lost.
```

```
-->
<events>
    <event id="0" name="Baro_Acquired" severity="ACTIVITY_HI" ...
       format_string="acquired">
        <comment>A notification on Barometer lock acquired</comment>
    </event>
    <event id="1" name="Baro_lost" severity="WARNING_HI" ...
       format_string="lock lost">
        <comment>A warning on Barometer lock lost</comment>
    </event>
</events>
```

E.2 CMakeLists

Listing E.5: CMakeLists.txt RPI

```
# 'RPI' Deployment:
#
# This sets up the build for the 'RPI' Application, including the custom
# components. In addition, it imports FPrime.cmake, which includes the ...
   core F Prime
# components.
#
# This file has several sections.
#
# 1. Header Section: define basic properties of the build
# 2. F prime core: includes all F prime core components, and build-system ...
   properties
# 3. Local subdirectories: contains all deployment specific directory ...
   additions
####

##
# Section 1: Basic Project Setup
#
# This contains the basic project information. Specifically, a cmake ...
   version and
# project definition.
##
project(RPI C CXX)
cmake_minimum_required(VERSION 3.13)
set(FPRIME_FRAMEWORK_PATH "${CMAKE_CURRENT_LIST_DIR}/..(path to Fprime ...
   Cmake" CACHE PATH "Location of F prime framework" FORCE)
set(FPRIME_PROJECT_ROOT "${CMAKE_CURRENT_LIST_DIR}/..(path to Fprime Code ...
   Cmake" CACHE PATH "Root path of F prime project" FORCE)

##
# Section 2: F prime Core
#
# This includes all of the F prime core components, and imports the ...
   make-system. F prime core
# components will be placed in the F-Prime binary subdirectory to keep ...
   them from
# colliding with deployment specific items.
##
include("${CMAKE_CURRENT_LIST_DIR}/../cmake/FPrime.cmake")
# NOTE: register custom targets between these two lines
include("${CMAKE_CURRENT_LIST_DIR}/../cmake/FPrime-Code.cmake")
```

```
# Note: when building a deployment outside of the F prime core ...
    directories, then the
# build root must be re-mapped for use with the standard build system ...
    components.
#
# In this way, the module names can be predicted as an offset from the ...
    (new) build
# root, without breaking the standard locations of F prime.
#
# Uncomment the following lines, and set them to the BUILD_ROOT of your ...
    deployment,
# which is typically one directory up from the CMakeLists.txt in the ...
    deployment dir.
#set(FPRIME_CURRENT_BUILD_ROOT "${CMAKE_CURRENT_LIST_DIR}/..")
#message(STATUS "F prime BUILD_ROOT currently set to: ...
    ${FPRIME_CURRENT_BUILD_ROOT}")

##
# Section 3: Components and Topology
#
# This section includes deployment specific directories. This allows use ...
    of non-
# core components in the topology, which is also added here.
##
# Add component subdirectories
add_fprime_subdirectory("${CMAKE_CURRENT_LIST_DIR}/RpiDemo/")

# Add Topology subdirectory
add_fprime_subdirectory("${CMAKE_CURRENT_LIST_DIR}/Top/")

#Add Barometer subdirectory
add_fprime_subdirectory("${CMAKE_CURRENT_LIST_DIR}/Barometer/")

set(SOURCE_FILES "${CMAKE_CURRENT_LIST_DIR}/Main.cpp")
set(MOD_DEPS ${PROJECT_NAME}/Top)

register_fprime_deployment()
```

Appendix F

Flashing OS to RPI Compute Module

To flash an OS to the Raspberry Pi Compute Module, the USBBOOT driver from Raspberry is needed. This can be found on their github page located here [102]. This must the be installed on the Ubuntu in this case, but also works out of the box in both Windows and Mac OS. When this is installed, the EMMC-boot on the Compute Module must be shorted to prevent the Pi from booting. This is shown in figure F.1.

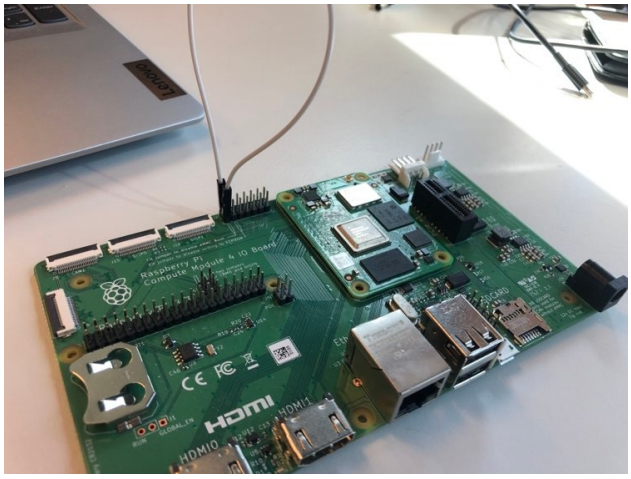


Figure F.1: Using a jumper wire to short the EMMC-boot

Now, go in to the usbboot directory on the host machine. In this case Ubuntu is used, so we open a terminal inside the directory and execute the make command to make a binary executable of the program. Connect a USB to the host machine, and the micro USB to the CM4 I/O board. Select the usb device under the device tree in the top left corner of the Ubuntu. Now write sudo ./rpiboot and the program locates the CM4 device and loads the boot folder. The device is now selectable in the Raspberry Pi Imager. The Imager is installed through the Ubuntu software app. When opening the imager, you get an image similar to the one in figure F.3

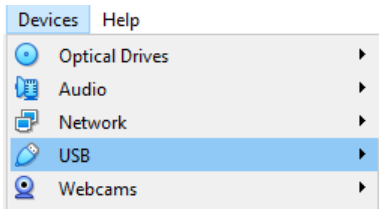


Figure F.2: Device tree of Ubuntu 20.04

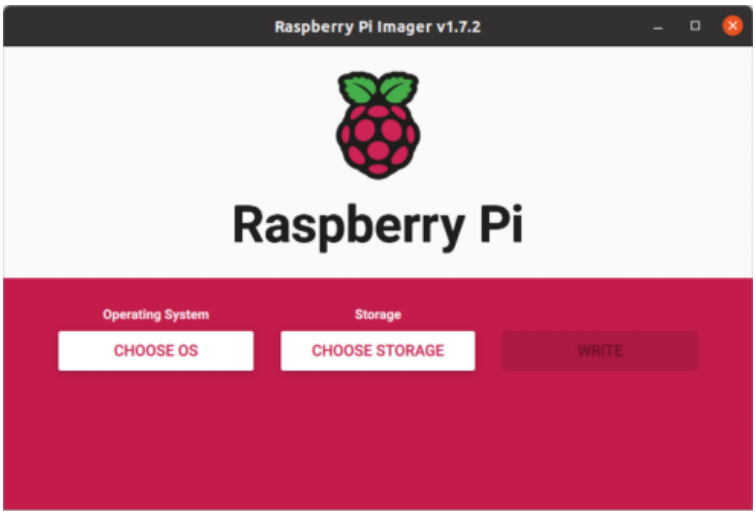


Figure F.3: Raspberry Pi Imager

From here the user can select what OS they want to be flashed. This can be an OS already in the Imager or for example a zip or img file containing a different OS. The user then chooses the Compute Module under "Choose storage" and from there press write. When the process is done, the OS has been flashed and the user can unmount the RPI from the computer. Now take the jumperwire for the EMMC-boot off, and power on the I/O board. If the user selected a desktop environment based OS, like Raspbian 64-bit, plug in the HDMI cable and the desktop should show and the Pi is now functioning with the OS.

Appendix G

Programming

G.1 Wiring Pi

Listing G.1: First code produced based off of previous bachelor [72]

```
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <string.h>
#include <wiringPi.h>
#include <wiringPiSPI.h>
#include <unistd.h>
#include <math.h>

// Headers
#include <bmi08x.h>

// MACROS
#define GRAVITY_EARTH (9.80665f)
#define PI (3.14159265f)
#define CS0 18
#define CS1 17
#define MISO 19
#define MOSI 20
#define SCLK 21
#define SPI_CHANNEL 1
#define BAUD 2000000
#define INTERRUPT_PIN 16
#define dt 0.01
// GLOBALS


// Accelerometer Addresses
const int REG_ACCEL_PWR_CTRL = 0x7D;
const int REG_ACCEL_FIFO_CONFIG_1 = 0x49;
const int ACCEL_CONF = 0x40;
const int ACCEL_RANGE = 0x41;
const int ACCEL_INT_CONF = 0x53;
const int ACCEL_IO_MAP = 0x58;

// Gyrometer Addresses
const int GYRO_PWR_CTRL = 0x11;
const int GYRO_FIFO_CONF = 0x3E;
const int GYRO_RANGE = 0x0F;
const int GYRO_BW = 0x00;
const int GYRO_INT_CTRL = 0x15;
```

```
const int GYRO_IO_MAP = 0x18;


// FUNCTIONS from Bosch
static float lsb_to_mps2(int16_t val, int8_t g_range, uint8_t bit_width);
static float lsb_to_dps(int16_t val, float dps, uint8_t bit_width);

// Function defenitions
void dataRdyInt(void);
void regConfig (const int setup, uint8_t *regAddr, uint8_t regVal, const ...
    uint16_t ms);
void ComplementaryFilter(float accData[3], float gyroData[3], float ...
    *pitch, float *roll);


// Variable declerations
    int setup, setupSPI;
    unsigned char buffer[100];
    volatile bool dataRdy;
    float x_a, y_a, z_a, x_g, y_g, z_g, readSPI; // Variables to hold ...
        sensordata and the readprocess
    float pitch = 0, roll = 0; // Variables for the complementary filter

int main()
{
    setup = wiringPiSetup();
    if(setup < 0){
        printf("WiringPi Setup failed!");
        return -1;
    }

    setupSPI = wiringPiSPISetup(SPI_CHANNEL, BAUD);

    if (setupSPI == -1){
        printf("Spi Setup failed!");
        return -1;
    }

    if (wiringPiISR(INTERRUPT_PIN, INT_EDGE_RISING, &dataRdyInt) < 0){
        printf("Unable to set up interrupt");
        return -1;
    }

    // Accelerometer Setup
    buffer[0] = (ACCEL_CONF << 1U) | 0U;
    regConfig(1, buffer, 0x08, 2); // ODR 100 HZ

    buffer[0] = (ACCEL_INT_CONF << 1U) | 0U;
    regConfig(1, buffer, 0x80, 2); // Accel data rdy int initilazer

    buffer[0] = (ACCEL_IO_MAP << 1U) | 0U;
    regConfig(1, buffer, 0x04, 2); // Data Ready Interrupt to INT pin 1

    buffer[0] = (ACCEL_RANGE << 1U) | 0U;
    regConfig(1, buffer, 0x03, 2); // Setting accel range to 16 G

    buffer[0] = (REG_ACCEL_PWR_CTRL << 1U) | 0U; // Send the Register ...
        Address in the first charcter of the buffer
    regConfig(1, buffer, 0x04, 50); // Waking the Accelerometer

    // Gyrometer Setup
```

```
    buffer[0] = (GYRO_RANGE << 1U) | 0U;
    regConfig(1, buffer, 0x02, 2); // Setting gyrometer range to +/- 500 dps

    buffer[0] = (GYRO_BW << 1U) | 0U;
    regConfig(1, buffer, 0x07, 2); // Gyro ODR to 100Hz

    while(!dataRdy){
        delay(10000);
    }

    buffer[0] = (0x12 << 1U) | 1U;
    buffer[1] = 0x55;
    buffer[2] = 0x00;
    readSPI = ((int16_t)wiringPiSPIDataRW(SPI_CHANNEL, buffer, 2));
    x_a = lsb_to_mps2(readSPI, 16, 16);

    buffer[0] = (0x14 << 1U) | 1U;
    buffer[1] = 0x55;
    buffer[2] = 0x00;
    readSPI = ((int16_t)wiringPiSPIDataRW(SPI_CHANNEL, buffer, 2));
    y_a = lsb_to_mps2(readSPI, 16, 16);

    buffer[0] = (0x16 << 1U) | 1U;
    buffer[1] = 0x55;
    buffer[2] = 0x00;
    readSPI = ((int16_t)wiringPiSPIDataRW(SPI_CHANNEL, buffer, 2));
    z_a = lsb_to_mps2(readSPI, 16, 16);

    buffer[0] = (0x02 << 1U) | 1U;
    buffer[1] = 0x55;
    buffer[2] = 0x00;
    readSPI = ((int16_t)wiringPiSPIDataRW(SPI_CHANNEL, buffer, 2));
    x_g = lsb_to_dps(readSPI, 16, 16);

    buffer[0] = (0x04 << 1U) | 1U;
    buffer[1] = 0x55;
    buffer[2] = 0x00;
    readSPI = ((int16_t)wiringPiSPIDataRW(SPI_CHANNEL, buffer, 2));
    y_g = lsb_to_dps(readSPI, 16, 16);

    buffer[0] = (0x06 << 1U) | 1U;
    buffer[1] = 0x55;
    buffer[2] = 0x00;
    readSPI = ((int16_t)wiringPiSPIDataRW(SPI_CHANNEL, buffer, 2));
    z_g = lsb_to_dps(readSPI, 16, 16);

    printf("Accel X: %f, Accel Y: %f, Accel Z: %f\n ", x_a, y_a, z_a);
    printf("Gyro X: %f, Gyro Y: %f, Gyro Z: %f\n", x_g, y_g, z_g);

    // Store values in an array for the complementary filter
   //  float accArray[3] = {x_a, y_a, z_a};
   // float gyroArray[3] = {x_g, y_g, z_g};
}

void dataRdyInt (void){
    dataRdy = 1;
}

void regConfig (const int channel, uint8_t *regAddr, uint8_t regVal, const ...
   uint16_t ms){
```

```
    uint8_t *ADDRESS = regAddr;
    printf("Data from Register %p\n", ADDRESS);

    wiringPiSPIDataRW(channel, &regVal, 1);
    sleep(ms);
}

static float lsb_to_mps2(int16_t val, int8_t g_range, uint8_t bit_width)
{
    float gravity;

    float half_scale = ((1 << bit_width) / 2.0f);

    gravity = (float)((GRAVITY_EARTH * val * g_range) / half_scale);

    return gravity;
}

/*!
 * @brief This function converts lsb to degree per second for 16 bit gyro at
 * range 125, 250, 500, 1000 or 2000dps.
 */
static float lsb_to_dps(int16_t val, float dps, uint8_t bit_width)
{
    float half_scale = ((float)(1 << bit_width) / 2.0f);

    return (dps / ((half_scale) + BMI08X_GYRO_RANGE_2000_DPS)) * (val);
}

void ComplementaryFilter(float accData[3], float gyroData[3], float ...
   *pitch, float *roll){
    float pitchAcc, rollAcc;

    // Integrating gyro data
    *pitch += (gyroData[0] / 65.536) * dt; //Angel x-axis
    *roll -= (gyroData[1]/ 65.536) * dt; //Angel y-axis

    pitchAcc = (atan2f(accData[1], accData[2]) * 180)/PI;
    *pitch = *pitch * 0.98 + pitchAcc * 0.02;

    rollAcc = (atan2f(accData[0], accData[2]) * 180)/PI;
    *roll = *roll * 0.98 + rollAcc * 0.02;

    printf("Pitch%4.2f, Roll%4.2f\n", *pitch, *roll);

}
```

G.2 Pigpio

Listing G.2: Pigpio Code

```
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <string.h>
#include <unistd.h>
#include <math.h>
#include <pigpio.h>
#include <time.h>

// Headers
#include <bmi08x.h>

// MACROS
#define GRAVITY_EARTH (9.80665f)
#define PI (3.14159265f)
#define CS0 18
#define CS1 17
#define MISO 19
#define MOSI 20
#define SCLK 21
#define SPI_CHANNEL 1
#define BAUD 2000000
#define INTERRUPT_PIN 16
#define dt 0.01


// Accelerometer Addresses
const int REG_ACCEL_PWR_CTRL = 0x04;
const int REG_ACCEL_FIFO_CONFIG_1 = 0x49;
const int ACCEL_CONF = 0x40;
const int ACCEL_RANGE = 0x41;
const int ACCEL_INT_CONF = 0x53;
const int ACCEL_IO_MAP = 0x58;

// Gyrometer Addresses
const int GYRO_PWR_CTRL = 0x11;
const int GYRO_FIFO_CONF = 0x3E;
const int GYRO_RANGE = 0x0F;
const int GYRO_BW = 0x00;
const int GYRO_INT_CTRL = 0x15;
const int GYRO_IO_MAP = 0x18;


// Function declerations

static float lsb_to_mps2(int16_t val, int8_t g_range, uint8_t bit_width);

static float lsb_to_dps(int16_t val, float dps, uint8_t bit_width);

// FUNCTIONS


void ComplementaryFilter(float accData[3], float gyroData[3], float ...
    *pitch, float *roll);
```

```
// Variable declerations
    int setup, setupSPI;
    unsigned char buffer[100];
    volatile bool dataRdy;
    float x_a, y_a, z_a, x_g, y_g, z_g, readSPI; // Variables to hold ...
       sensordata and the readprocess
    float pitch = 0, roll = 0; // Variables for the complementary filter

int main()
{

    if (gpioInitialise() < 0){
        printf("Pigpio Init failed!");
    }
    else{
        printf("Pigpio OK");

    }

    gpioWrite(CS1, 0);
    sleep(1);
    int setup = bbSPIOpen(CS1, MISO, MOSI, SCLK, BAUD, PI_SPI_FLAGS_MODE(0));

    printf("%d\n", setup);
    char t1[] = {(BMI08X_REG_ACCEL_CHIP_ID << 1U) | 0, 0};
    unsigned char inBuf[8];
    int tx = bbSPIXfer(CS1, t1, (char*)inBuf, 2);

    printf("%d", tx);

    char t2[] = {(BMI08X_REG_ACCEL_PWR_CTRL << 1U | 0), ...
       REG_ACCEL_PWR_CTRL, 0};
    tx = bbSPIXfer(CS1, t2, (char*)inBuf, 2);

    char t3[] = {(BMI08X_REG_ACCEL_CONF << 1U | 0), ACCEL_CONF};
    tx = bbSPIXfer(CS1, t3, (char*)inBuf, 2);

    char t4[] = {(BMI08X_REG_ACCEL_INT1_IO_CONF << 1U | 0), ACCEL_INT_CONF};
    tx = bbSPIXfer(CS1, t4, (char*)inBuf, 2);

    char t5[] = {(BMI08X_REG_ACCEL_INT1_INT2_MAP_DATA << 1U | 0), ...
       ACCEL_IO_MAP};
    tx = bbSPIXfer(CS1, t5, (char*)inBuf, 2);

    char t6[] = {(BMI085_ACCEL_RANGE_16G << 1U | 0), ACCEL_RANGE};
    tx = bbSPIXfer(CS1, t6, (char*)inBuf, 2);

    char t7[] = {(BMI08X_GYRO_RANGE_500_DPS << 1U | 0), GYRO_RANGE};
    tx = bbSPIXfer(CS0, t7, (char*)inBuf, 2);

    char t8[] = {(BMI08X_GYRO_BW_32_ODR_100_HZ << 1U | 0), GYRO_BW};
    tx = bbSPIXfer(CS0, t8, (char*)inBuf, 2);

    while (gpioInitialise() > 0) {
        char t9[] = {(BMI08X_REG_GYRO_X_LSB << 1U | 1), ...
           BMI08X_REG_GYRO_Y_LSB, BMI08X_REG_GYRO_Z_LSB};
        unsigned char inBuf1[8];

        tx = bbSPIXfer(CS1, t9, (char*)inBuf1, 4);
```

```
        float x_val = (int16_t) ((inBuf1[1]&3)<<8);
        float y_val = (int16_t) ((inBuf1[2]&3)<<8);
        float z_val = (int16_t) ((inBuf1[3]&3)<<8);

        printf("X:%f, Y:%f, Z:%f\n", x_val, y_val, z_val);
    }
    //printf("Accel X: %f, Accel Y: %f, Accel Z: %f\n ", x_a, y_a, z_a);
    //printf("Gyro X: %f, Gyro Y: %f, Gyro Z: %f\n", x_g, y_g, z_g);

    // Store values in an array for the complementary filter
  //  float accArray[3] = {x_a, y_a, z_a};
   // float gyroArray[3] = {x_g, y_g, z_g};

   bbSPIClose(CS1);

return 0;
}




static float lsb_to_mps2(int16_t val, int8_t g_range, uint8_t bit_width)
{
    float gravity;

    float half_scale = ((1 << bit_width) / 2.0f);

    gravity = (float)((GRAVITY_EARTH * val * g_range) / half_scale);

    return gravity;
}

/*!
 * @brief This function converts lsb to degree per second for 16 bit gyro at
 * range 125, 250, 500, 1000 or 2000dps.
 */
static float lsb_to_dps(int16_t val, float dps, uint8_t bit_width)
{
    float half_scale = ((float)(1 << bit_width) / 2.0f);

    return (dps / ((half_scale) + 100)) * (val);
}

void ComplementaryFilter(float accData[3], float gyroData[3], float ...
   *pitch, float *roll){
    float pitchAcc, rollAcc;

    // Integrating gyro data
    *pitch += (gyroData[0] / 65.536) * dt; //Angel x-axis
    *roll -= (gyroData[1]/ 65.536) * dt; //Angel y-axis

    pitchAcc = (atan2f(accData[1], accData[2]) * 180)/PI;
    *pitch = *pitch * 0.98 + pitchAcc * 0.02;

    rollAcc = (atan2f(accData[0], accData[2]) * 180)/PI;
    *roll = *roll * 0.98 + rollAcc * 0.02;

    printf("Pitch%4.2f, Roll%4.2f\n", *pitch, *roll);

}
```

G.3 Spidev & IOCTL

Listing G.3: Spidev code

```
#include <stdio.h>
#include <stdint.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/ioctl.h>
#include <gpiod.h>
#include <linux/spi/spidev.h>
#include "gpiolib.h"


// MACROS
#define GRAVITY_EARTH (9.80665f)
#define PI (3.14159265f)
#define CS0 18
#define CS1 17
#define MISO 19
#define MOSI 20
#define SCLK 21
#define SPI_CHANNEL 1
#define BAUD 2000000
#define INTERRUPT_PIN 16
#define dt 0.01

// Accelerometer Addresses
const int REG_ACCEL_PWR_CTRL = 0x7D;
const int REG_ACCEL_FIFO_CONFIG_1 = 0x49;
const int ACCEL_CONF = 0x40;
const int ACCEL_RANGE = 0x41;
const int ACCEL_INT_CONF = 0x53;
const int ACCEL_IO_MAP = 0x58;

// Gyrometer Addresses
const int GYRO_PWR_CTRL = 0x11;
const int GYRO_FIFO_CONF = 0x3E;
const int GYRO_RANGE = 0x0F;
const int GYRO_BW = 0x10;
const int GYRO_INT_CTRL = 0x15;
const int GYRO_IO_MAP = 0x18;

// Global variables
float x_a, y_a, z_a, x_g, y_g, z_g;
int f_spi;


// Function declerations
static struct spi_ioc_transfer xfer;



int set_gpio_pin(int gpio_num, int direction){
    // export the GPIO pin to gpiolib
    gpio_export(gpio_num);

    // Setting direction as output
```

```
    gpio_direction(gpio_num, 1);

    return 0;
}

int write_to_gpio(int gpio_num, int value){

    gpio_write(gpio_num, value);

    return 0;
}

int init_spi(){
    int val = 0;
    int iSPIMode = SPI_MODE_0;
    int spi_freq = 2000000;
    f_spi = open("/dev/spidev1.1", O_RDWR);

    if(f_spi < 0){
        perror("Error initialising the SPI device\n");
        return -1;
    }

    val = ioctl(f_spi, SPI_IOC_WR_MODE, &iSPIMode);
    if (val < 0){
        perror("Error setting the SPI mode\n");
        return -1;
    }

    val = ioctl(f_spi, SPI_IOC_WR_MAX_SPEED_HZ, &spi_freq);
    if (val < 0){
        perror("Error setting the SPI bus speed\n");
        return -1;
    }

    memset(&xfer, 0, sizeof (xfer));
    xfer.speed_hz = spi_freq;
    xfer.cs_change = 0;
    xfer.delay_usecs = 2;
    xfer.bits_per_word = 8;

    return 0;
}

void spi_write(uint8_t *pstring, uint8_t length){
    int ret_val = 0;
    ret_val = write_to_gpio(CS1, 0);

    if (ret_val != 0){
        perror("Failed to write the GPIO pin\n");
        exit(EXIT_FAILURE);
    }
    xfer.rx_buf = 0;
    xfer.tx_buf = (unsigned long)pstring;
    xfer.len = length;

    ret_val = ioctl(f_spi, SPI_IOC_MESSAGE(1), &xfer);

    ret_val = write_to_gpio(CS1, 1);
    if (ret_val != 0){
        perror("Failed to write the GPIO pin\n");
```

```
        exit(EXIT_FAILURE);
    }

    return;
}


int main()
{
    int val = 0;
    uint8_t buf[16];


    val = init_spi();
    if (val == -1){
        perror("Failed to initialise SPI bus\n");
        return -1;
    }

    set_gpio_pin(CS0, GPIO_OUT); // CS to accelerometer set tp output
    write_to_gpio(CS1, 1); // To switch the accelerometer to SPI, it must ...
       detect a rising edge on the CS pin

    // Configuring the IMU

    /* Accelerometer*/
    buf[0] = (ACCEL_CONF << 1U) | 0U;
    buf[1] = 0x55;
    buf[2] = 0x08; // ODR 100Hz
    spi_write(buf, 3);

    buf[0] = (ACCEL_INT_CONF << 1U) | 0U;
    buf[1] = 0x55;
    buf[2] = 0x0A; // Initializing the INT pin 1
    spi_write(buf, 3);

    buf[0] = (ACCEL_IO_MAP << 1U) | 0U;
    buf[1] = 0x55;
    buf[2] = 0x04; // IO map mapped to int pin 1
    spi_write(buf, 3);

    buf[0] = (ACCEL_RANGE << 1U)| 0U;
    buf[1] = 0x55;
    buf[2] = 0x03; // Accel range set to 16G
    spi_write(buf, 3);

    /* Gyroscope */
    buf[0] = (GYRO_RANGE << 1U) | 0U;
    buf[1] = 0x55;
    buf[2] = 0x02; // Setting gyro range to +/- 500 dps
    spi_write(buf, 3);

    buf[0] = (GYRO_BW << 1U) | 0U;
    buf[1] = 0x55;
    buf[2] = 0x07; // ODR Frequency to 100 Hz
    spi_write(buf, 3);

    /* Powering on the Accelerometer*/
    buf[0] = (REG_ACCEL_PWR_CTRL << 1U) | 0U;
    buf[1] = 0x55;
    buf[2] = 0x04; // This starts the accelerometer from sleep mode
```

```
    spi_write(buf, 3);
    sleep(50/1000000); // Sleep for 50 microseconds

    while(val != -1){
    /* Start the accelerometer readings */
    buf[0] = (0x12 << 1U) | 1U;
    buf[1] = 0x55;
    buf[2] = 0x12;
    spi_write(buf, 3);
    x_a = (int16_t)buf[2];
    printf("X_A:%c ", buf[1]);
    buf[0] = (0x14 << 1U) | 1U;
    buf[1] = 0x55;
    buf[2] = 0x14;
    spi_write(buf, 3);
    y_a = (int16_t)&buf[2];

    buf[0] = (0x16 << 1U) | 1U;
    buf[1] = 0x55;
    buf[2] = 0x16;
    spi_write(buf, 3);
    z_a = (int16_t)buf[2];

    printf("Accel X:%d, Accel Y:%f, Accel Z: %f\n", buf[2], y_a, z_a);

    /* Start the gyroscope readings*/
    buf[0] = (0x02 << 1U) | 1U;
    buf[1] = 0x55;
    buf[2] = 0x02;
    x_g = (int16_t)buf[2];

    buf[0] = (0x04 << 1U) | 1U;
    buf[1] = 0x55;
    buf[2] = 0x04;
    y_g = (int16_t)buf[2];

    buf[0] = (0x06 << 1U) | 1U;
    buf[1] = 0x55;
    buf[2] = 0x06;
    z_g = (int16_t)buf[2];

    printf("Gyro X:%f, Gyro Y:%f, Gyro Z:%f\n", x_g, y_g, z_g);
    }

    return 0;
}
```

G.3.1 Shuttle board code

Listing G.4: Using the Wiring Pi SPI library

```
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <string.h>
#include <pigpio.h>
#include <wiringPi.h>
#include <wiringPiSPI.h>
#include <unistd.h>
#include <math.h>
#include <pthread.h>

#define GRAVITY_EARTH (9.80665f)
#define PI (3.14159265f)
#define CS0 8
#define CS1 7
#define MISO 19
#define MOSI 20
#define SCLK 21
#define SPI_CHANNEL 0
#define BAUD 2000000

// Accelerometer Addresses
const int REG_ACCEL_PWR_CTRL = 0x7D;
const int REG_ACCEL_FIFO_CONFIG_1 = 0x49;
const int ACCEL_CONF = 0x40;
const int ACCEL_RANGE = 0x41;
const int ACCEL_INT_CONF = 0x53;
const int ACCEL_IO_MAP = 0x58;

// Gyrometer Addresses
const int GYRO_PWR_CTRL = 0x11;
const int GYRO_FIFO_CONF = 0x3E;
const int GYRO_RANGE = 0x0F;
const int GYRO_BW = 0x00;
const int GYRO_INT_CTRL = 0x15;
const int GYRO_IO_MAP = 0x18;

void dataRdy(void);
_Bool rdy;

unsigned char buf[100];
float x_a, y_a, z_a, x_g, y_g, z_g;
int main()
{
    if(wiringPiSetup() < 0){
        printf("WiringPi Setup failed!\n");
    }
    else {
        printf("Wiring Pi Setup OK!\n");
    }

    if (wiringPiSetupGpio() < 0){
        printf("Failed to open GPIO\n");
    }

    if (wiringPiSPISetupMode(SPI_CHANNEL, BAUD, 0) < 0){
        printf("Failed to set SPI-Mode\n");
```

```
    }

    digitalWrite(CS0, 1);
    delayMicroseconds(10);

    if(wiringPiISR(16, INT_EDGE_RISING, &dataRdy)){
        printf("Failed to set interrupt\n");
    }

    unsigned char buf[3] = {(ACCEL_CONF << 1U) | 0U, 0x08, 0x00};
     // ODR 100Hz
    wiringPiSPIDataRW(SPI_CHANNEL, buf, sizeof(buf));
    delayMicroseconds(2);

    unsigned char buf2[3] = {(ACCEL_INT_CONF << 1U) | 0U, 0x0A, 0x00};
     // Initializing the INT pin 1
    wiringPiSPIDataRW(SPI_CHANNEL, buf2, sizeof (buf2));
    delayMicroseconds(2);

    unsigned char buf3[3] = {(ACCEL_IO_MAP << 1U) | 0U, 0x04, 0x00};
     // IO map mapped to int pin 1
    wiringPiSPIDataRW(SPI_CHANNEL, buf3, sizeof(buf3));
    delayMicroseconds(2);

    unsigned char buf4[3] = {(ACCEL_RANGE << 1U)| 0U, 0x03, 0x00};
    // Accel range set to 16G
    wiringPiSPIDataRW(SPI_CHANNEL, buf4, sizeof (buf4));
    delayMicroseconds(2);

    /* Gyroscope */
    unsigned char buf5[3] = {(GYRO_RANGE << 1U) | 0U, 0x02, 0x00};
    // Setting gyro range to +/- 500 dps
    wiringPiSPIDataRW(SPI_CHANNEL, buf5, sizeof (buf5));
    delayMicroseconds(2);

    unsigned char buf6[3] = {(GYRO_BW << 1U) | 0U, 0x07, 0x00};
    // ODR Frequency to 100 Hz
    wiringPiSPIDataRW(SPI_CHANNEL, buf6, sizeof(buf6));
    delayMicroseconds(2);

    /* Powering on the Accelerometer*/

    unsigned char buf7[3] = {(REG_ACCEL_PWR_CTRL << 1U) | 0U, 0x04, 0x00};
    // This starts the accelerometer from sleep mode
    wiringPiSPIDataRW(SPI_CHANNEL, buf7, sizeof (buf7));
    delayMicroseconds(500);


    while (1) {
        unsigned char buffer[3] = {0x12 << 1U | 1U, 0x00, 0x00};

        wiringPiSPIDataRW(SPI_CHANNEL, buffer, sizeof (buffer));
        x_a = buffer[2];
        printf("Accel X:%f\n", x_a);
        delay(2);
        printf("Accel Xbuf: %s", &buffer[2]);
    }
    return 0;
}

void dataRdy(void){
```

```
    rdy = 1;
}
```

Listing G.5: Using the IOCTL & spidev libraries

```
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <string.h>
#include <unistd.h>
#include <math.h>
#include <pthread.h>
#include <sys/ioctl.h>
#include <linux/spi/spidev.h>
#include "gpiolib.h"
#include <fcntl.h>
#include <gpiod.h>
#include <wiringPi.h>

#define CS0 8
#define CS1 7
#define MODE 0
#define SPEED 2000000

// ----- GLOBALS------
int f_spi, g_spi;
static struct spi_ioc_transfer xfer;
static struct spi_ioc_transfer gfer;

// Accelerometer Addresses
const int REG_ACCEL_PWR_CTRL = 0x7D;
const int REG_ACCEL_FIFO_CONFIG_1 = 0x49;
const int ACCEL_CONF = 0x40;
const int ACCEL_RANGE = 0x41;
const int ACCEL_INT_CONF = 0x53;
const int ACCEL_IO_MAP = 0x58;

// Gyrometer Addresses
const int GYRO_PWR_CTRL = 0x11;
const int GYRO_FIFO_CONF = 0x3E;
const int GYRO_RANGE = 0x0F;
const int GYRO_BW = 0x10;
const int GYRO_INT_CTRL = 0x15;
const int GYRO_IO_MAP = 0x18;

// -----Functions ------

int set_gpio(int pin_num){
    // export the GPIO pin to gpiolib
    wiringPiSetup();
    pinMode(pin_num, OUTPUT);

    // Setting direction as output/input


    return 0;
}

int write_gpio(int pin_num, char value){
    digitalWrite(pin_num, value);
```

```
}

int spi_accel_init(){
    int val = 0;
    int iSPIMode = SPI_MODE_0;
    int spi_freq = 2000000;
    f_spi = open("/dev/spidev0.0", O_RDWR);

    if(f_spi < 0){
        perror("Error initialising the SPI device\n");
        return -1;
    }

    val = ioctl(f_spi, SPI_IOC_WR_MODE, &iSPIMode);
    if (val < 0){
        perror("Error setting the SPI mode\n");
        return -1;
    }

    val = ioctl(f_spi, SPI_IOC_WR_MAX_SPEED_HZ, &spi_freq);
    if (val < 0){
        perror("Error setting the SPI bus speed\n");
        return -1;
    }

    memset(&xfer, 0, sizeof (xfer));
    xfer.speed_hz = spi_freq;
    xfer.cs_change = 0;
    xfer.delay_usecs = 2;
    xfer.bits_per_word = 8;

    return 0;
}

int spi_gyro_init(){
    int val = 0;
    int iSPIMode = SPI_MODE_0;
    int spi_freq = 2000000;
    g_spi = open("/dev/spidev0.1", O_RDWR);

    if(g_spi < 0){
        perror("Error initialising the SPI device\n");
        return -1;
    }

    val = ioctl(g_spi, SPI_IOC_WR_MODE, &iSPIMode);
    if (val < 0){
        perror("Error setting the SPI mode\n");
        return -1;
    }

    val = ioctl(g_spi, SPI_IOC_WR_MAX_SPEED_HZ, &spi_freq);
    if (val < 0){
        perror("Error setting the SPI bus speed\n");
        return -1;
    }

    memset(&gfer, 0, sizeof (gfer));
    gfer.speed_hz = spi_freq;
    gfer.cs_change = 0;
    gfer.delay_usecs = 2;
```

```
    gfer.bits_per_word = 8;

    return 0;
}

void spi_write_accel(uint8_t *pstring, uint8_t length){
    int ret_val = 0;
    digitalWrite(CS0, LOW);

    xfer.rx_buf = 0;
    xfer.tx_buf = (unsigned long)pstring;
    xfer.len = length;

    ret_val = ioctl(f_spi, SPI_IOC_MESSAGE(1), &xfer);

    digitalWrite(CS0, HIGH);
    return;
}

void spi_write_gyro(uint8_t *pstring, uint8_t length){
    int ret_val = 0;
    digitalWrite(CS1, LOW);

    gfer.rx_buf = 0;
    gfer.tx_buf = (unsigned long)pstring;
    gfer.len = length;

    ret_val = ioctl(g_spi, SPI_IOC_MESSAGE(1), &gfer);

    digitalWrite(CS1, HIGH);
    return;
}


int main()
{
    int val = 0;
    uint8_t buf[3];

    set_gpio(CS0); // Accel CS as output
    digitalWrite(CS0, HIGH); // Accel CS-pin to HIGH to signal SPI


    val = spi_accel_init();
    if (val == -1){
        perror("Failed to initialise SPI bus\n");
        return -1;
    }

    val = spi_gyro_init();
    if (val == -1){
        perror("Failed to initialise SPI bus\n");
        return -1;
    }

    buf[0] = (ACCEL_CONF << 1U) | 0U;
    buf[1] = 0x08;
    buf[2] = 0x00; // ODR 100Hz
    spi_write_accel(buf, 3);
    sleep(2/1000000);
```

```
    buf[0] = (ACCEL_INT_CONF << 1U) | 0U;
    buf[1] = 0x0A;
    buf[2] = 0x00; // Initializing the INT pin 1
    spi_write_accel(buf, 3);
    sleep(2/1000000);

    buf[0] = (ACCEL_IO_MAP << 1U) | 0U;
    buf[1] = 0x04;
    buf[2] = 0x00; // IO map mapped to int pin 1
    spi_write_accel(buf, 3);
    sleep(2/1000000);

    buf[0] = (ACCEL_RANGE << 1U)| 0U;
    buf[1] = 0x03;
    buf[2] = 0x00; // Accel range set to 16G
    spi_write_accel(buf, 3);
    sleep(2/1000000);

    /* Gyroscope */
    buf[0] = (GYRO_RANGE << 1U) | 0U;
    buf[1] = 0x02;
    buf[2] = 0x00; // Setting gyro range to +/- 500 dps
    spi_write_gyro(buf, 3);
    sleep(2/1000000);

    buf[0] = (GYRO_BW << 1U) | 0U;
    buf[1] = 0x07;
    buf[2] = 0x00; // ODR Frequency to 100 Hz
    spi_write_gyro(buf, 3);
    sleep(2/1000000);

    /* Powering on the Accelerometer*/
    buf[0] = (REG_ACCEL_PWR_CTRL << 1U) | 0U;
    buf[1] = 0x04;
    buf[2] = 0x00; // This starts the accelerometer from sleep mode
    spi_write_accel(buf, 3);
    sleep(50/1000000); // Sleep for 50 microseconds

    while(val != -1){

        buf[0] = (0x12 << 1U) | 1U;
        buf[1] = 0x00;
        buf[2] = 0x00;
        spi_write_accel(buf, 3);
        printf("Accel X:%d\n", buf[2]);
        sleep(2/1000000);


        buf[0] = (0x14 << 1U) | 1U;
        buf[1] = 0x00;
        buf[2] = 0x00;
        spi_write_accel(buf, 3);
        printf("Accel Y:%d\n", buf[2]);
        sleep(2/1000000);


        buf[0] = (0x16 << 1U) | 1U;
        buf[1] = 0x00;
        buf[2] = 0x00;
        spi_write_accel(buf, 3);
        printf("Accel Z:%d\n", buf[2]);
```

```
        sleep(2/1000000);

    }



return 0;

}
```

Listing G.6: Using the Wiring Pi I2C library

```
#include <stdio.h>
#include <stdint.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <wiringPi.h>
#include <wiringPiI2C.h>
#include <math.h>

// Accelerometer Addresses
const int REG_ACCEL_PWR_CTRL = 0x7D;
const int REG_ACCEL_FIFO_CONFIG_1 = 0x49;
const int ACCEL_CONF = 0x40;
const int ACCEL_RANGE = 0x41;
const int ACCEL_INT_CONF = 0x53;
const int ACCEL_IO_MAP = 0x58;

// Gyrometer Addresses
const int GYRO_PWR_CTRL = 0x11;
const int GYRO_FIFO_CONF = 0x3E;
const int GYRO_RANGE = 0x0F;
const int GYRO_BW = 0x10;
const int GYRO_INT_CTRL = 0x15;
const int GYRO_IO_MAP = 0x18;

void configReg(const int fd, const int regAddr, const int regVal, const ...
   uint16_t ms);
void msleep(uint16_t ms);
float x, y, z, xg, yg, zg;


int main()
{
    if (wiringPiSetup() < 0){
        printf("Failed to set up Wiring Pi\n");
        return -1;
    }

    int fd_acc = wiringPiI2CSetup(0x19);
    if (fd_acc == -1){
        printf("Accelerometer not found!\n");
        return -1;
    }

    int fd_gyr = wiringPiI2CSetup(0x68);
    if (fd_gyr == -1) {
        printf("Gyroscope not found!\n");
        return -1;
```

```
    }
    configReg(fd_acc, REG_ACCEL_PWR_CTRL, 0x00, 2); // Power of the ...
       accelerometer to ensure it is woken with the correct ...
       configurations each time

    // ----Basic configurations----
    configReg(fd_acc, ACCEL_CONF, 0xA8, 2);
    configReg(fd_acc, ACCEL_INT_CONF, 0x0A, 2);
    configReg(fd_acc, ACCEL_IO_MAP, 0x04, 2);
    configReg(fd_acc, ACCEL_RANGE, 0x01, 2);

    configReg(fd_gyr, GYRO_RANGE, 0x02, 2);
    configReg(fd_gyr, GYRO_BW, 0x87, 2);

    configReg(fd_acc, REG_ACCEL_PWR_CTRL, 0x04, 50);

    while(1){
      x = ((int16_t)wiringPiI2CReadReg16(fd_acc, 0x12)) / (5460/9.81);
      y = ((int16_t)wiringPiI2CReadReg16(fd_acc, 0x14))/ (5460/9.81);
      z = ((int16_t)wiringPiI2CReadReg16(fd_acc, 0x16))/ (5460/9.81);
      //printf("X.%f, Y:%f, Z:%f\n", x, y, z);
      float accPitch = (180/3.141592) * atan2f(y, z);
      float accRoll = (180/3.141592) * atan2f(x,z);
      printf("Pitch:%f, Roll:%f\n", accPitch, accRoll);
      delay(50);
      }
}

void configReg(const int fd, const int regAddr, const int regVal, const ...
   uint16_t ms){
    wiringPiI2CWriteReg8(fd, regAddr, regVal);
    msleep(ms);
}

void msleep(uint16_t ms){
    useconds_t uSec = 1000*ms;
    usleep(uSec);
}
```

G.4 SPI-communication between Raspberry Pi and ATMega

G.4.1 Master code for the Raspberry Pi

Listing G.7: Master side code for the SPI-communication

```
#include <iostream>
#include <wiringPiSPI.h>
#define SPI_CHANNEL 0
#define SPI_CLOCK_SPEED 1000000
int main(int argc, char **argv)
{
    int fd = wiringPiSPISetupMode(SPI_CHANNEL, SPI_CLOCK_SPEED, 0);
    if (fd == -1) {
        std::cout << "Failed to init SPI communication.\n";
        return -1;
    }
    std::cout << "SPI communication successfully setup.\n";

    unsigned char buf[2] = { 23, 0 };
    wiringPiSPIDataRW(SPI_CHANNEL, buf, 2);
    std::cout << "Data returned: " << +buf[1] << "\n";
    return 0;
}
```

G.4.2 Slave code for the Arduino Nano

Listing G.8: Slave side code for the SPI-communication

```
#include <SPI.h>
void setup() {
  // have to send on master in, *slave out*
  pinMode(MISO, OUTPUT);
  // turn on SPI in slave mode
  SPCR |= _BV(SPE);
  // turn on interrupts
  SPI.attachInterrupt();
}
// SPI interrupt routine
ISR (SPI_STC_vect)
{
  byte c = SPDR;
  SPDR = c+10;
}  // end of interrupt service routine (ISR) for SPI
void loop () { }
```

G.5 IMU data from Raspberry Pi to Arduino

Listing G.9: Master code for the RPI

```
#include <wiringPiI2C.h>
#include <wiringPiSPI.h>
#include <stdlib.h>
#include <stdio.h>
#include <wiringPi.h>
#include <string.h>
#include <stdint.h>
#include <unistd.h>
#include <math.h>

#define Device_Address 0x68 /*Device Address/Identifier for MPU9250*/

#define CHANNEL 0
#define SPEED 1000000
#define PI 3.14159265
#define DT 0.05
#define PWR_MGMT_1   0x6B
#define SMPLRT_DIV   0x19
#define CONFIG       0x1A
#define GYRO_CONFIG  0x1B
#define INT_ENABLE   0x38
#define ACCEL_XOUT_H 0x3B
#define ACCEL_YOUT_H 0x3D
#define ACCEL_ZOUT_H 0x3F
#define GYRO_XOUT_H  0x43
#define GYRO_YOUT_H  0x45
#define GYRO_ZOUT_H  0x47

int fd;

void MPU9250_Init(){

    wiringPiI2CWriteReg8 (fd, SMPLRT_DIV, 0x07);    /* Write to sample ...
       rate register */
    wiringPiI2CWriteReg8 (fd, PWR_MGMT_1, 0x01);    /* Write to power ...
       management register */
    wiringPiI2CWriteReg8 (fd, CONFIG, 0);       /* Write to Configuration ...
       register */
    wiringPiI2CWriteReg8 (fd, GYRO_CONFIG, 24); /* Write to Gyro ...
       Configuration register */
    wiringPiI2CWriteReg8 (fd, INT_ENABLE, 0x01);    /*Write to interrupt ...
       enable register */

    }
short read_raw_data(int addr){
    short high_byte,low_byte,value;
    high_byte = wiringPiI2CReadReg8(fd, addr);
    low_byte = wiringPiI2CReadReg8(fd, addr+1);
    value = (high_byte << 8) | low_byte;
    return value;
}

void ms_delay(int val){
    int i,j;
    for(i=0;i<=val;i++)
        for(j=0;j<1200;j++);
```

```
}

int main(){

    float Acc_x,Acc_y,Acc_z;
    float Gyro_x,Gyro_y,Gyro_z;
    float Ax=0, Ay=0, Az=0;
    float Gx=0, Gy=0, Gz=0;
    fd = wiringPiI2CSetup(Device_Address);   /*Initializes I2C with device ...
       Address*/

    int fd1 = wiringPiSPISetupMode(CHANNEL, SPEED, 0); /* Initializing SPI */

    if (fd1 == -1) { printf("SPI failed\n");}

    MPU92500_Init();                         /* Initializes the IMU */

    while(1)
    {
        /*Read raw value of Accelerometer and gyroscope*/
        Acc_x = read_raw_data(ACCEL_XOUT_H);
        Acc_y = read_raw_data(ACCEL_YOUT_H);
        Acc_z = read_raw_data(ACCEL_ZOUT_H);

        Gyro_x = read_raw_data(GYRO_XOUT_H);
        Gyro_y = read_raw_data(GYRO_YOUT_H);
        Gyro_z = read_raw_data(GYRO_ZOUT_H);

        /* Divide raw value by sensitivity scale factor */
        Ax = 9.81*(Acc_x/16384.0); // +-2g sensitivity factor
        Ay = 9.81*(Acc_y/16384.0);
        Az = 9.81*(Acc_z/16384.0);

        Gx = Gyro_x/131; //250 deg/s sensitivity factor
        Gy = Gyro_y/131;
        Gz = Gyro_z/131;


        float init = sqrt((Ax*Ax) + (Az*Az));
        float init1 = sqrt((Ay*Ay)+(Az*Az));
        float accPitch = -(atan2f(Ay,Az))*(180/PI);
        float rollAcc = (atan2f(-Ax, init1))*(180/PI);

        printf("Pitch:%f, Roll X:%f\n", accPitch, rollAcc);

        uint8_t *pitchBuf = (uint8_t *)&accPitch;
        wiringPiSPIDataRW(CHANNEL, pitchBuf, sizeof(pitchBuf));

    }
    return 0;
}
```

Listing G.10: Slave code for the Arduino

```
#include <SPI.h>
volatile byte count = 0; // In Arduino, byte and uint8_t are reffered to ...
   as the same
volatile bool init; // Variable for initializing and stopping a read operation
float inVal; // Variable to store the incoming byte as a float
uint8_t *ptr = (uint8_t *)&inVal; // Making a pointer to the float ...
   variable to avoid data type mixing
```

```
void setup() {
  Serial.begin(9600);
  pinMode(MISO, OUTPUT); // Setting the MISO as output

  SPCR |= _BV(SPE); // This command enables slave mode on the Arduino
  SPCR |= _BV(SPIE);
  init = false;
  SPI.attachInterrupt(); // Enabeling interrupts

}

ISR (SPI_STC_vect) // Setting up interrupt routine
{
  ptr[count++]= SPDR; // Storing the recieved data from the SPI Data ...
     Register to the pointer which point to the float variable

  if (count == 2){
    init = true; // When the bytes are recieved, we set the print ...
       initialization to true
  }
}
void loop() {
  if (init = true) {

    Serial.print("Acceleration X:"); Serial.println(value); // Printing ...
       the value variable, which contains the bytes sent by the RPI
    count = 0; // Resetting the byte counter
    init = false; // Putting init to false to indicate we want to read the ...
       next transfer
    }
}
```

G.6 PID-control and Motor Mixing Algorithm

Listing G.11: PID algorithm

```
#include <PID_v1.h>

#include <Wire.h>
#include <MPU9250_WE.h>
#include <Servo.h>


#define MPU6500_ADDR 0x68
MPU6500_WE myMPU6500 = MPU6500_WE(MPU6500_ADDR);

Servo Motor1;
Servo Motor2;
Servo Motor3;
Servo Motor4;

// ------ PID ------
double pitchSetpoint, pitchInput, pitchOutput;
double rollSetpoint, rollInput, rollOutput;
double Kp = 1, Ki = 0.01, Kd = 0;

PID myPID(&pitchInput, &pitchOutput, &pitchSetpoint, Kp, Ki, Kd, REVERSE); ...
   // Motor 1+2 is CCW
```

```
PID myPID2(&rollInput, &rollOutput, &rollSetpoint, Kp, Ki, Kd, REVERSE); ...
    // Motor 3+4 is CW

// Globals
int speed;

void setup() {
  Serial.begin(115200);
  Wire.begin();

  delay(200);

  Motor1.attach(3, 1000, 2000);
  Motor2.attach(9, 1000, 2000);
  Motor3.attach(10, 1000, 2000);
  Motor4.attach(11, 1000, 2000);

  delay(50);
  myMPU6500.autoOffsets();
  myMPU6500.enableGyrDLPF();
  myMPU6500.setGyrDLPF(MPU6500_DLPF_6); // Low pass filter setting for ...
      lowest noise
  myMPU6500.setSampleRateDivider(5);
  myMPU6500.setGyrRange(MPU6500_GYRO_RANGE_250);
  myMPU6500.setAccRange(MPU6500_ACC_RANGE_2G);
  myMPU6500.enableAccDLPF(true);
  myMPU6500.setAccDLPF(MPU6500_DLPF_6);

  delay(200);

  pitchSetpoint = 0.0;
  rollSetpoint = 0.0;
  myPID.SetMode(AUTOMATIC);
  myPID2.SetMode(AUTOMATIC);

}

void loop() {
  xyzFloat gValue = myMPU6500.getGValues();
  xyzFloat gyr = myMPU6500.getGyrValues();
  xyzFloat angels = myMPU6500.getAngles();

  float pitch = -1 * myMPU6500.getPitch();
  float roll = -1 * myMPU6500.getRoll();


  Serial.print("Pitch :"); Serial.println(pitch);
  Serial.print("Roll:"); Serial.println(roll);

  speed = analogRead(A0);
  speed = map(speed, 0, 1023, 0, 180); // PWM converted to servo signal ...
      for the motors
  pitchInput = pitch;
  rollInput = roll;
  int pulse, pulse1, pulse2, pulse3;
  myPID.Compute();
  myPID2.Compute();
  pulse = speed + pitchOutput - rollOutput;
  pulse1 = speed - pitchOutput + rollOutput;
  pulse2 = speed + pitchOutput + rollOutput;
  pulse3 = speed - pitchOutput - rollOutput;
```

```
  Motor1.write(pulse);
  Motor2.write(pulse1);
  Motor3.write(pulse2);
  Motor4.write(pulse3);

}
```

Appendix H

Ordering history

Figure H.1 shows every component that needed to be ordered in. The figure also gives information about which suppliers, manufactures and what kind of components are used in this project. The right side of the figure also shows the pricing of each object.

Quantity	Type of components	Product Number	Supplier	Manufacturer	Price per 1 unit	Total
10	**Barometer**	bmp384	Digi-Key	**Bosch Sensortec**	kr39,32	kr393,15
100	**Low Noise LDO**	AP2138N-3.3TRG1	RS-components	**DiodesZetex**	**kr1,94**	**kr193,80**
1	**Clear Plastic Sheet**	334-6444	RS-components	**RS PRO**	kr257,87	kr257,87
1	Solder Paste	SMD291AX	Farnell	CHIP QUIK	kr317,24	kr317,24
1	Dispenser, Syringe Tips, 30 Pa	SMDTA30	Farnell	CHIP QUIK	kr188,24	kr188,24
20	LDO Voltage Regulators	AP2138N-3.3TRG1	Mouser	**Diodes Incorporated**	kr2,81	kr56,20
15	PCB Header	501331-0307	Mouser	Molex	kr4,32	kr64,80
15	Pico-Clasp	501330-0300	Mouser	Molex	kr1,83	kr27,45
100	Lighting Connectors (cut-strip)	501334-0000	Mouser	Molex	kr0,46	kr46,00
10	Optical sensor	VL53L5CXV0GC/1	Digi-key	STMicroelectronics	kr85,49	kr772,28
10	Line driver	NLV17SZ07DFT2G	Digi-Key	onsemi	kr4,49	kr36,19
1	Expansion Module	134-6482	RS components	Digilent		**kr305,27**
20	Line driver	SN74LVC1G07DCKR	Farnell	Texas instrument	**kr3,09**	kr61,80
100	SMD chip Resitor	RC0603JR-0722RL	Farnell	Yageo	**kr0,07**	kr7,08
3	Kakute F4 All-in-one V2	11031	Holybro	Holybro		362,32
4	Barometric sensor	BMP384	Digi-key	Bosch-sensortec	kr50,46	kr201,84
4	IMU	BMI085	Digi-key	Bosch-sensortec		
2	Microcontroller	ATMEGA32U4RC-MUR	Digi-key	Microchip Technology	kr50,90	kr101,80
2	Compute Module 4 I/O Board	SC0326	Farnell	Raspberry Pi	kr306,60	kr613,20

Figure H.1: Components needed to order in

Appendix I

Bill of materials PCB

Id	Designator	Quantity	Size and name
1	C9,C8,C6,C7	4	100nF
2	R5,R9,R8	3	100K
3	R10,R11,R12,R13	4	0 Ohm
4	U1	1	SN74LVC1G07DCKR
5	C3	1	3.3nF
6	D2	1	LED Red
7	R14,R15	2	22 Ohm
8	J3	1	Connector I2C JST_SM06B
9	JP1,JP2,JP3	3	Solder Jumper
10	U3	1	MIC5019YFT-TR
11	J5	1	TPS62133RGTR
12	J7,J2,J9,J8	4	Connector PWM Pin-header 3x1mm
13	C5	1	4.7uF
14	U8	1	AP2138N-3.3TRG1
15	C11,C13,C2,C10,C12	5	0.1uF
16	U6	1	ATMEGA32U4
17	U7	1	BMP384
18	D1	1	LED Green

Id	Designator	Quantity	Size and name
19	J1	1	Connector micro-USB
20	R6	1	261K
21	J6	1	Hole-thru pin
22	R4,R3	2	2.2K
23	R2,R1	2	1k
24	REF**,VCC_AS	2	VCC_PAD
25	REF**,GND_AS	2	GND_PAD
26	SW1	1	Dip-switch
27	C4	1	22uF
28	J10	1	Hole-thru pin
29	L1	1	2.2uH
30	J4	1	Connector micro-USB
31	R7	1	51K
32	C1	1	10uF
33	Q1	1	Solder Pad
34	U2	1	TPD2EUSB30
35	Module1	1	CM4
36	U5	1	BMI085

Figure I.1: Bill of materials for the PCB

Appendix J

Building instructions

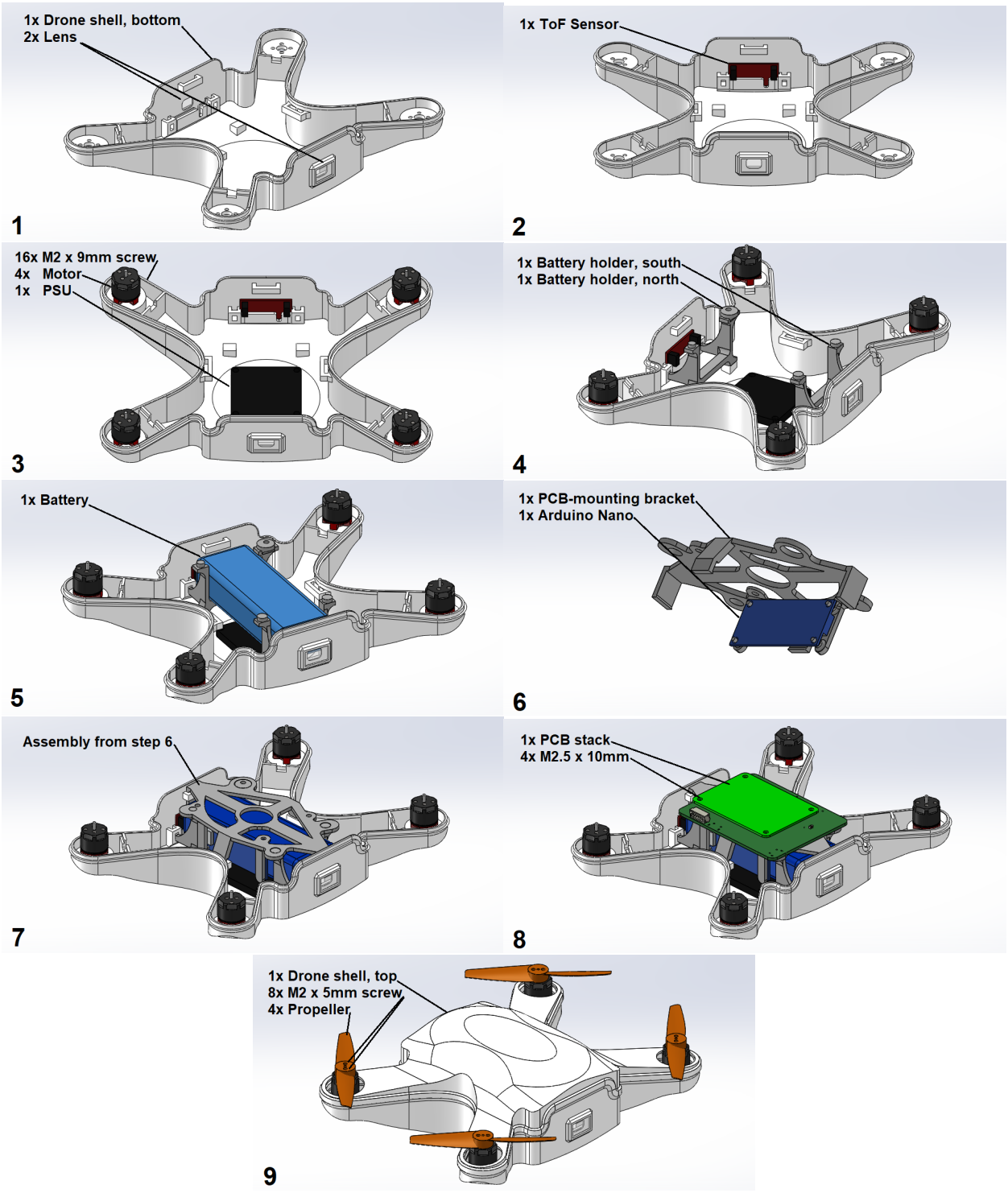


Figure J.1: Mounting manual with steps

ITEM NO.	PART NAME	DESCRIPTION	QTY.
1	Drone shell, bottom	Outerpart of the drone	1
2	Drone shell, top	Outerpart of the drone	1
3	Lens	Lens to protect ToF sensor	2
4	ToF sensor	VL53L5CX on Sparkfun breakout board	1
5	Motor	EMAX RS1106 Micro Brushless Motor	4
6	PSU	Holybro PM06 V, power supply unit	1
7	Battery holder, north	Battery holder	1
8	Battery holder, south	Battery holder	1
9	Battery	Turnigy LiPo 2s 1600 mAh 20-30 C	1
10	PCB-mounting bracket	Holder of the PCB stack	1
11	Arduino Nano	Arduino Nano, with ATMEGA328Pchip	1
12	PCB stack	V3 of selfdeveloped PCB card, with a CM4 on top.	1
13	Propeller	2 bladed propeller	4
14	M2 x 9mm screw	Screw for mounting the motor to the drone body	16
15	M2.5 x 10mm screw	Screw for mounting the PCB stack to the bracket	4
16	M2 x 5mm screw	Screw for mounting the propeller to the motor	8

Figure J.2: Bill of materials, drone structure.

Bibliography

[1] Khan Academy. *TCP explained*. URL: https://www.khanacademy.org/computing/computers-and-internet/xcae6f4a7ff015e7d:the-internet/xcae6f4a7ff015e7d:transporting-packets/a/transmission-control-protocol--tcp. (accessed: 18.05.2022).

[2] ActLab. *Crazyflie 2.0 swarm*. URL: https://www.youtube.com/watch?v=gH1Fcf597So. (accessed: 05.05.2022).

[3] Adafruit. *Differences between AVR microproccesors*. URL: https://learn.adafruit.com/how-to-choose-a-microcontroller?view=all. (accessed: 05.05.2022).

[4] ArchLinux. *Realtime kernel patchset*. URL: https://wiki.archlinux.org/title/Realtime_kernel_patchset. (accessed: 16.05.2022).

[5] Arduino. *Arduino Leonardo*. URL: https://www.arduino.cc/en/main/arduinoBoardLeonardo. (accessed: 11.05.2022).

[6] Arduino. *Arduino Nano*. URL: http://store.arduino.cc/products/arduino-nano. (accessed: 11.05.2022).

[7] Arduino. *Servo*. URL: https://www.arduino.cc/reference/en/libraries/servo/. (accessed: 13.05.2022).

[8] Ardupilot. *Mission Planner Home*. URL: https://ardupilot.org/planner/. (accessed: 16.05.2022).

[9] Ardupilot. *Motor order diagrams*. URL: https://ardupilot.org/copter/docs/connect-escs-and-motors.html. (accessed: 16.05.2022).

[10] ARM. *ARM Cortex A17*. URL: https://dbpedia.org/page/ARM_Cortex-A17. (accessed: 11.05.2022).

[11] astroesteban/nasa. *INSTALL.md*. URL: https://github.com/nasa/fprime/blob/devel/docs/INSTALL.md. (accessed: 17.01.2022).

[12] ASUS. *ASUS*. URL: https://www.bhphotovideo.com/c/product/1418508-REG/asus_90me0031_m0aay0_tinker_board_s_motherboard.html. (accessed: 11.05.2022).

[13] ASUS. *Best Raspberry Pi Alternatives*. URL: http://dlcdnet.asus.com/pub/ASUS/mb/Embedded_IPC/TinkerBoard_S/E13446_Tinker_Board_S_UM_WEB.pdf. (accessed: 11.05.2022).

[14] Atmel. *Atmel ATmega32U4*. URL: https://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-7766-8-bit-AVR-ATmega16U4-32U4_Datasheet.pdf. (accessed: 22.03.2022).

[15] Robotics Back-end. *Raspberry Pi (master) Arduino Uno (slave) SPI communication with WiringPi*. URL: https://roboticsbackend.com/raspberry-pi-master-arduino-uno-slave-spi-communication-with-wiringpi/. (accessed: 11.05.2022).

[16] Mojtaba Bagherzadeh et al. "Analyzing a decade of Linux system calls." In: *Empirical Software Engineering* 23.3 (2018), pp. 1519–1551.

[17] Circuit Basics. *BASICS OF THE I2C COMMUNICATION PROTOCOL*. URL: https://www.circuitbasics.com/basics-of-the-i2c-communication-protocol/. (accessed: 04.05.2022).

[18] Circuit Basics. *BASICS OF THE SPI COMMUNICATION PROTOCOL*. URL: https://www.circuitbasics.com/basics-of-the-spi-communication-protocol/. (accessed: 04.05.2022).

[19] Steve Bate. *ChibiOS-RPi*. URL: https://github.com/steve-bate/ChibiOS-RPi. (accessed: 16.05.2022).

[20] Brett Beauregard. *Arduino PID Library*. URL: https://playground.arduino.cc/Code/PIDLibrary/. (accessed: 13.05.2022).

[21] bitcraze. *Crazyflie 2.1*. URL: https://www.bitcraze.io/products/crazyflie-2-1/. (accessed: 05.05.2022).

[22] Bosch. *BMI085*. URL: https://www.bosch-sensortec.com/products/motion-sensors/imus/bmi085/. (accessed: 02.05.2022).

[23] Bosch. *BMI085 Datasheet*. URL: https://www.bosch-sensortec.com/media/boschsensortec/downloads/datasheets/bst-bmi085-ds001.pdf. (accessed: 22.03.2022).

[24] Bosch. *BMI088*. URL: https://www.bosch-sensortec.com/products/motion-sensors/imus/bmi088/. (accessed: 02.05.2022).

[25] Bosch. *BMI088 Datasheet*. URL: https://www.bosch-sensortec.com/media/boschsensortec/downloads/datasheets/bst-bmi088-ds001.pdf. (accessed: 02.05.2022).

[26] Bosch. *BMP384 Barometric Pressure Sensor Datasheet*. URL: https://www.bosch-sensortec.com/media/boschsensortec/downloads/datasheets/bst-bmp384-ds003.pdf. (accessed: 28.03.2022).

[27] BoschSensortec. *BMI08x-Sensor-API*. URL: https://github.com/BoschSensortec/BMI08x-Sensor-API. (accessed: 10.05.2022).

[28] BoschSensortec. *BMP3-Sensor-API*. URL: https://github.com/BoschSensortec/BMP3-Sensor-API. (accessed: 10.05.2022).

[29] Tone Braadland. *barometer i Store norske leksikon på snl.no*. URL: https://snl.no/barometer. (accessed: 04.03.2022).

[30] Christian Cawley. *Raspberry Pi Zero 2 W: Good Upgrade, but Needs More RAM*. URL: https://www.makeuseof.com/raspberry-pi-zero-2-w-review/. (accessed: 11.05.2022).

[31] chipquik. *Solder Paste No-Clean Sn63/Pb37 in 5cc Syringe 15g*. URL: http://www.chipquik.com/datasheets/SMD291AX.pdf. (accessed: 16.05.2022).

[32] Sierra Circuits. *What is the use of decoupling capasitors*. URL: https://www.protoexpress.com/blog/decoupling-capacitor-use/. (accessed: 04.05.2022).

[33] RS-Components. *Raspberry Pi Compute Module 4 (CM4) with WiFi 8GB, 32GB Flash*. URL: https://no.rs-online.com/web/p/raspberry-pi/2064847. (accessed: 11.05.2022).

[34] CompTIA. *What is NAT?* URL: https://www.comptia.org/content/guides/what-is-network-address-translation. (accessed: 16.05.2022).

[35] Computersalg. *Asus Tinker Board S*. URL: https://www.computersalg.no/i/7896622/asus-tinker-board-s-r2-0-enkeltbrettsdatamaskin-rockchip-rk3288-cg-w-ram-2-gb-flash-16-gb-802-11b-g-n-bluetooth-4-2-edr?utm_source=prisjaktNo&utm_medium=prisjaktNoLINK&utm_campaign=prisjaktNo. (accessed: 11.05.2022).

[36] TE Connectivity. *MS5611-01BA03*. URL: https://www.te.com/commerce/DocumentDelivery/DDEController?Action=showdoc&DocId=Data+Sheet%7FMS5611-01BA03%7FB3%7Fpdf%7FEnglish%7FENG_DS_MS5611-01BA03_B3.pdf%7FCAT-BLPS0036. (accessed: 30.03.2022).

[37] DLNWARE. *Connecting Multiple Slave Devices*. URL: https://dlnware.com/dll/Connecting-multiple-slave-devices. (accessed: 04.05.2022).

[38] Downey, Allen. *The little book of semaphores*. Green Tea Press, 2008.

[39] drmatt. *cm4 dead - any recovery options?* URL: https://forums.raspberrypi.com/viewtopic.php?t=314038. (accessed: 10.05.2022).

[40] E-tinkers. *Do you know Arduino? – SPI and Arduino SPI Library*. URL: https://www.e-tinkers.com/2020/03/do-you-know-arduino-spi-and-arduino-spi-library/. (accessed: 11.05.2022).

[41] ElectronicWings. *MPU6050 (Accelerometer+Gyroscope) Interfacing with Raspberry Pi*. URL: https://www.electronicwings.com/raspberry-pi/mpu6050-accelerometergyroscope-interfacing-with-raspberry-pi. (accessed: 11.05.2022).

[42] ELRPROCUS. *Gyroscope Sensor Working and Its Applications*. URL: https://www.elprocus.com/gyroscope-sensor/. (accessed: 16.05.2022).

[43] EmbeddedCraft. *Using SPI in Embedded Linux Part 3: Code Explanation, LED Matrix Display Controller*. URL: https://www.youtube.com/watch?v=6x4VaLGWX90&t=383s&ab_channel=EmbeddedCraft. (accessed: 11.05.2022).

[44] ChibiOS EmbeddedWare. *ChibiOS free embedded RTOS - ChibiOS Homepage*. URL: https://www.chibios.org/dokuwiki/doku.php. (accessed: 16.05.2022).

[45] Carsten Emde. *ptsematest(8)*. URL: https://man.archlinux.org/man/community/rt-tests/ptsematest.8.en. (accessed: 07.04.2022).

[46] Carsten Emde. *sigwaittest(8)*. URL: https://man.archlinux.org/man/sigwaittest.8. (accessed: 07.04.2022).

[47] Carsten Emde. *svsematest(8)*. URL: https://man.archlinux.org/man/svsematest.8. (accessed: 07.04.2022).

[48] Farnell. *RPI ZERO W V2*. URL: https://no.farnell.com/raspberry-pi/rpi-zero-w-v2/raspberry-pi-kit-64bit-arm-cortex/dp/3838499?src=raspberrypi. (accessed: 11.05.2022).

[49] Linux Foundation. *RT-Tests*. URL: https://wiki.linuxfoundation.org/realtime/documentation/howto/tools/rt-tests. (accessed: 24.03.2022).

[50] FreeRTOS. *FreeRTOS™ - Real-time operating system for microcontrollers*. URL: https://www.freertos.org/. (accessed: 16.05.2022).

[51] Thomas Gleixner. *Cyclictest*. URL: https://man.archlinux.org/man/cyclictest.8. (accessed: 07.04.2022).

[52] RJ Hardt. *Using Metrology to Measure and Enhance the Performance of a Positioning System*. URL: https://www.aerotech.com/using-metrology-to-measure-and-enhance-the-performance-of-a-positioning-system/. (accessed: 02.05.2022).

[53] Gordon Henderson. *Wiring Pi*. URL: http://wiringpi.com/. (accessed: 10.05.2022).

[54] Diodes Inc. *ULTRA LOW QUIESCENT CURRENT CMOS LDO*. URL: https://no.mouser.com/datasheet/2/115/DIOD_S_A0008668704_1-2513012.pdf. (accessed: 25.04.2022).

[55] Texas Instruments. *Step Down Converter*. URL: https://www.ti.com/lit/ds/symlink/tps62133.pdf?HQS=dis-mous-null-mousermode-dsf-pf-null-wwe&ts=1651570439105&ref_url=https%253A%252F%252Fno.mouser.com%252F. (accessed: 03.05.2022).

[56] Insider Intelligence. *Drone technology uses and applications for commercial, industrial and military drones in 2021 and the future*. URL: https://www.businessinsider.com/drone-technology-uses-applications?r=US&IR=T. (accessed: 05.05.2022).

[57] InvenSense. *MPU-9250 Register Map and Descriptions*. URL: https://3cfeqx1hf82y3xcoull08ihx-wpengine.netdna-ssl.com/wp-content/uploads/2017/11/RM-MPU-9250A-00-v1.6.pdf. (accessed: 13.05.2022).

[58] Iotrends. *Raspberry Pi Compute Module 4 – A Complete Guide*. URL: https://www.iottrends.tech/blog/raspberry-pi-compute-module-4-everything-you-need-to-know/. (accessed: 11.05.2022).

[59] Preeti Jain. *Magnetometers*. URL: https://www.engineersgarage.com/magnetometers/. (accessed: 16.05.2022).

[60] joan2937. *PigPio*. URL: https://abyz.me.uk/rpi/pigpio/index.html. (accessed: 10.05.2022).

[61] Martin Bjaadal Økter Jørgen Borge Martin Sauar Wad. "Drone Arena for Reinforcement Learning." In: (2022), 150<.

[62] Jan Petter Ottesen Jørgen Mikal Benum Simon Hus and Ola Christoffer Våge. "Modulbasert drone tilpasset autonom bruk i Motion Capture system." In: (2018), p. 160.

[63] JST. *GH Connector*. URL: https://www.jst-mfg.com/product/pdf/eng/eGH.pdf. (accessed: 28.04.2022).

[64] kdoren. *Linux 5.10.90-rt*. URL: https://github.com/kdoren/linux. (accessed: 03.03.2022).

[65] Kernel.org. *RT-kernel patch*. URL: https://mirrors.edge.kernel.org/pub/linux/kernel/projects/rt/. (accessed: 15.02.2022).

[66] kernel.org. *Linux 5.10.90*. URL: https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/commit/?h=linux-5.10.y&id=d3e491a20d152e5fba6c02a38916d63f982d98a5. (accessed: 01.03.2022).

[67] Kicad. *Introduction to Kicad*. URL: https://docs.kicad.org/#_getting_started. (accessed: 09.05.2022).

[68] Kingbright. *2.0x1.25mm SMD CHIP LED LAMP*. URL: https://www.farnell.com/datasheets/1701218.pdf. (accessed: 03.05.2022).

[69] Adrian Kingsley-Hughes. *Best Raspberry Pi Alternatives*. URL: https://www.zdnet.com/article/best-raspberry-pi-alternative/. (accessed: 11.05.2022).

[70] kjellco. *Nvidia Jetson*. URL: https://www.kjell.com/no/produkter/elektro-og-verktoy/utviklerkit/nvidia-jetson-nano-developer-kit-enkortsdatamaskin-for-bl.a.-maskinlaering-p88103. (accessed: 11.05.2022).

[71] Manon Kok et al. "Calibration of a magnetometer in combination with inertial sensors." In: *2012 15th International Conference on Information Fusion*. 2012, pp. 787–793. URL: https://ieeexplore.ieee.org/abstract/document/6289882?casa_token=rh7C8SV4gz8AAAAA:9FDQUy2pWBxGgPvYkBMz-u78mPXsylXlfCDbApJfn33ttOGF_QfriN_ytrE9xdK5ihWAmS-ATQ.

[72] Marius Egeland Kristoffer Hansen Kruithof. "Modulær drone for svermeksperimenter i UiA Motion Lab." In: (2019), p. 161.

[73] Steen Larsen and Ben Lee. *Chapter Two - Survey on System I/O Hardware Transactions and Impact on Latency, Throughput, and Other Factors*. Ed. by Ali Hurson. Vol. 92. Advances in Computers. Elsevier, 2014, pp. 67–104. DOI: https://doi.org/10.1016/B978-0-12-420232-0.00002-7. (accessed: 10.05.2022).

[74] Store Norske Leksikon. *polymetylmetakrylat*. URL: https://snl.no/polymetylmetakrylat. (accessed: 08.04.2022).

[75] LeMaRiva. *N-Queens-problem*. URL: https://github.com/lemariva/N-Queens-Problem.git. (accessed: 07.04.2022).

[76] LeMariva. *Raspberry Pi 4B: Real-Time System using Preempt-RT (kernel 4.19.y)*. URL: https://lemariva.com/blog/2019/09/raspberry-pi-4b-preempt-rt-kernel-419y-performance-test. (accessed: 20.03.2022).

[77] Lovdata. *Forskrift om luftfartøy som ikke har fører om bord mv*. URL: https://lovdata.no/dokument/SF/forskrift/2015-11-30-1404#KAPITTEL_2. (accessed: 08.04.2022).

[78] Luftfarttilsynet. *Hvilke regler gjelder for droner under 250 gram??* URL: https://luftfartstilsynet.no/droner/faq---droner---samlete-sporsmal/sporsmal-1/regler-for-droner-under-250-gram/. (accessed: 18.05.2022).

[79] Jan-Henrik Skogstad Martin Maeland Martin D. Hermansen. "Automatisk utskytningssystem for dronesverm." In: (2021), p. 216.

[80] Microchip. *ATmega328p*. URL: https://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-7810-Automotive-Microcontrollers-ATmega328P_Datasheet.pdf. (accessed: 05.05.2022).

[81] Microchip. *AtTiny85*. URL: https://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-7669-ATtiny25-45-85-Appendix-B-Automotive-Specification-at-1.8V_Datasheet.pdf. (accessed: 05.05.2022).

[82] Microsoft. *What is .NET Framework?* URL: https://dotnet.microsoft.com/en-us/learn/dotnet/what-is-dotnet-framework. (accessed: 16.05.2022).

[83] Molex. *Molex Cut Strip Datasheet*. URL: https://no.mouser.com/datasheet/2/276/2/5013340000_CRIMP_TERMINALS-179230.pdf. (accessed: 20.04.2022).

[84] Molex. *Molex Female Pin Connector*. URL: https://www.molex.com/webdocs/datasheets/pdf/en-us/5013300300_CRIMP_HOUSINGS.pdf. (accessed: 20.04.2022).

[85] Molex. *Molex Male Pin Connector*. URL: https://no.mouser.com/datasheet/2/276/2/5013310307_PCB_HEADERS-179250.pdf. (accessed: 20.04.2022).

[86] Saptarishi Mondal. *Difference between Spinlock and Semaphore*. URL: https://www.geeksforgeeks.org/difference-between-spinlock-and-semaphore/. (accessed: 16.05.2022).

[87] NASA. *F' A Flight Software and Embedded systems Framework*. URL: https://nasa.github.io/fprime/. (accessed: 02.05.2022).

[88] NASA. *F' GPS Tutorial*. URL: https://nasa.github.io/fprime/Tutorials/GpsTutorial/Tutorial.html. (accessed: 30.03.2022).

[89] Jet Propulsion Laboratory / NASA. *CUBESATS AND SMALLSATS*. URL: https://www.jpl.nasa.gov/topics/cubesats. (accessed: 02.05.2022).

[90] nicepng. *Windows Mac Linux logo*. URL: https://www.nicepng.com/ourpic/u2w7w7y3e6t4w7t4_windows-mac-linux-logo/. (accessed: 07.04.2022).

[91] NuttX. *Detailed Platform Support*. URL: https://nuttx.apache.org/docs/latest/introduction/detailed_support.html. (accessed: 16.05.2022).

[92] Nvidia. *Nvidia Jetson*. URL: https://elinux.org/Jetson_Nano#Other. (accessed: 11.05.2022).

[93] Nvidia. *NVIDIA Jetson Nano System-on-Module*. URL: https://developer.download.nvidia.com/assets/embedded/secure/jetson/Nano/docs/JetsonNano_DataSheet_DS09366001v1.0.pdf?FfbTRxPSWE5pgFxAXXChlpWSIvYPwTFv-2fAs8hYhjxNOxuYVortnP1WZEdPw5WRFbMKPv-aBV9aIfwJs3eE8cZL6CKPDBlbZiVnz3np65Bp1Mrspz16h_v8-CfgkoC6Qg_s2boV_T7afl1Q&t=eyJscyI6ImdzZW8iLCJsc2QiOiJodHRwczpcL1wvd3d3Lmdvb2dsZS5jb21cLyJ9. (accessed: 11.05.2022).

[94] NXP. *I2C-bus specifications*. URL: https://www.nxp.com/docs/en/user-guide/UM10204.pdf. (accessed: 23.03.2022).

[95] Omega. *PID explained*. URL: https://www.omega.co.uk/prodinfo/how-does-a-pid-controller-work.html. (accessed: 16.05.2022).

[96] ROS Organization. *Understanding ROS Nodes*. URL: http://wiki.ros.org/ROS/Tutorials/UnderstandingNodes. (accessed: 02.05.2022).

[97] Andrea Paterniani. *linux/spidev.h at master*. URL: https://github.com/spotify/linux/blob/master/include/linux/spi/spidev.h. (accessed: 11.05.2022).

[98] Anders Persson. "The Coriolis Effect." In: *History of Meteorology* 2 (2005), pp. 1–24. URL: https://journal.meteohistory.org/index.php/hom/article/view/30.

[99] Raspberry Pi. *Linux Kernel Patching*. URL: https://www.raspberrypi.com/documentation/computers/linux_kernel.html#configuring-the-kernel. (accessed: 15.02.2022).

[100] Raspberry Pi. *Raspberry Pi Compute Module 4 Datasheet*. URL: https://datasheets.raspberrypi.com/cm4/cm4-datasheet.pdf. (accessed: 23.03.2022).

[101] Raspberry Pi. *Raspberry Pi Compute Module 4 IO Board Datasheet*. URL: https://datasheets.raspberrypi.com/cm4io/cm4io-datasheet.pdf. (accessed: 23.03.2022).

[102] Raspberry Pi. *raspberrypi/usbboot*. URL: https://github.com/raspberrypi/usbboot. (accessed: 10.02.2022).

[103] Pieter-Jan. *Reading a IMU Without Kalman: The Complementary Filter*. URL: https://www.pieter-jan.com/node/11. (accessed: 16.05.2022).

[104] Pixhawk. *Flying 101 PX4 User Guide*. URL: https://docs.px4.io/v1.10/zh/flying/basic_flying.html. (accessed: 16.05.2022).

[105] protosupplies. *GY-63 MS5611 Pressure Temperature Sensor Module*. URL: https://protosupplies.com/product/gy-63-ms5611-pressure-temperature-sensor-module/. (accessed: 30.03.2022).

[106] Ritesh Ranjan. *What is a Framework in Programming Why You Should Use One*. URL: https://www.netsolutions.com/insights/what-is-a-framework-in-programming/. (accessed: 16.05.2022).

[107] raspberrypi. *Firmware / Bump to 5.10.90*. URL: https://github.com/raspberrypi/firmware/commit/318e33734963f7163d4747aaddb81b40239ba73c. (accessed: 01.03.2022).

[108] Low-dropout regulator. *Low-dropout regulator*. URL: https://en.wikipedia.org/wiki/Low-dropout_regulator. (accessed: 22.03.2022).

[109] ROS. *ROS: Home*. URL: https://www.ros.org/. (accessed: 16.05.2022).

[110] Steven Rostedt and John Kacur. *RT-MIGRATE-TEST(8)*. URL: https://man.archlinux.org/man/community/rt-tests/rt-migrate-test.8.en. (accessed: 07.04.2022).

[111] Atheer L Salih et al. "Modelling and PID controller design for a quadrotor unmanned air vehicle." In: *2010 IEEE International Conference on Automation, Quality and Testing, Robotics (AQTR)*. Vol. 1. IEEE. 2010, pp. 1–5.

[112] Paul G Savage. *What Do Accelerometers Measure?* URL: http://www.strapdownassociates.com/Accels%20Measure.pdf. (accessed: 16.05.2022).

[113] Bosch Sensortec. *BMI085 Shuttle Board 3.0 flyer*. URL: https://no.mouser.com/datasheet/2/783/bst_bmi085_sf000-2486535.pdf. (accessed: 13.05.2022).

[114] Sparkfun. *Adding WiFi to the NVIDIA Jetson*. URL: https://learn.sparkfun.com/tutorials/adding-wifi-to-the-nvidia-jetson/all. (accessed: 11.05.2022).

[115] Sparkfun. *SparkFun Qwiic Mini ToF Imager - VL53L5CX*. URL: https://www.sparkfun.com/products/19013. (accessed: 14.05.2022).

[116] P. Srisuresh and K. Egevang. "Traditional IP Network Address Translator (Traditional NAT)." In: *RFC 3022* (). DOI: 10.17487/RFC3022. URL: https://www.rfc-editor.org/rfc/rfc3022. (accessed: 16.05.2022).

[117] Staaker. *Best Drone Controller 2022: Top Brands Reviewed*. URL: https://staaker.com/best-drone-controller/. (accessed: 05.05.2022).

[118] John A Stankovic and Raj Rajkumar. "Real-time operating systems." In: *Real-Time Systems* 28.2-3 (2004), pp. 237–253.

[119] STMicroelectronics. *vl53l5cx-datasheet*. URL: https://www.st.com/en/imaging-and-photonics-solutions/vl53l5cx.html. (accessed: 12.05.2022).

[120] ES systems. *MEMS Capacitive vs Piezoresistive Pressure Sensors – What are their differences?* URL: https://esenssys.com/capacitive-piezoresistive-pressure-sensors-differences/. (accessed: 16.05.2022).

[121] Prodigy Technovations. *I2C vs SPI*. URL: https://prodigytechno.com/i2c-vs-spi/. (accessed: 23.03.2022).

[122] TechTarget. *Port Address Translation (PAT)*. URL: https://www.techtarget.com/searchnetworking/definition/Port-Address-Translation-PAT. (accessed: 16.05.2022).

[123] Henrik H. Togba et al. *Flight Controller Design and Imple-mentation for UiA Drone Project*. URL: https://www.youtube.com/watch?v=hROYKfuhZZc&ab_channel=HenrikHansenTogba. (accessed: 16.05.2022).

[124] Linus Torvalds. *gpiolib.h & gpiolib.c*. URL: https://github.com/torvalds/linux/tree/master/drivers/gpio. (accessed: 11.05.2022).

[125] Linus Torvalds. *ioctl(2) — Linux manual page*. URL: https://man7.org/linux/man-pages/man2/ioctl.2.html. (accessed: 11.05.2022).

[126] Linus Torvalds. *linux/ioctl.h at master*. URL: https://github.com/torvalds/linux/blob/master/arch/alpha/include/uapi/asm/ioctl.h. (accessed: 11.05.2022).

[127] Tutorialspoint. *Embedded Systems - Interrupts*. URL: https://www.tutorialspoint.com/embedded_systems/es_interrupts.htm. (accessed: 16.05.2022).

[128] Aaron Vinet Judd. Griffin and Levente Polyák. *ArchLinux*. URL: https://archlinux.org/. (accessed: 07.04.2022).

[129] K. C. Wang. *General Purpose Embedded Operating Systems*. Springer International Publishing, 2017, pp. 265–327. ISBN: 978-3-319-51517-5. DOI: 10.1007/978-3-319-51517-5_8. URL: https://doi.org/10.1007/978-3-319-51517-5_8.

[130] "D.C Whalley". ""A simplified reflow soldering process model"." In: *"Journal of Materials Processing Technology"* "150"."1" (2004), "134–144". ISSN: "0924-0136". DOI: "https://doi.org/10.1016/j.jmatprotec.2004.01.029". URL: https://www.sciencedirect.com/science/article/pii/S0924013604000858.

[131] Randy | Whatsabyte. *What are threads in a processor*. URL: https://whatsabyte.com/blog/processor-threads/. (accessed: 16.05.2022).

[132] Wikipedia. *I2C*. URL: https://en.wikipedia.org/wiki/I%C2%B2C. (accessed: 23.03.2022).

[133] Wikipedia. *Serial Peripheral Interface*. URL: https://en.wikipedia.org/wiki/Serial_Peripheral_Interface. (accessed: 23.03.2022).

[134] Wikipedia. *Time-of-flight camera*. URL: https://en.wikipedia.org/wiki/Time-of-flight_camera. (accessed: 04.04.2022).

[135] Wollewald. *MPU9250$_W$E*. URL: https://github.com/wollewald/MPU9250_WE. (accessed: 13.05.2022).

[136] Mingyang Zhang et al. "Which Is the Best Real-Time Operating System for Drones? Evaluation of the Real-Time Characteristics of NuttX and ChibiOS." In: *2021 International Conference on Unmanned Aircraft Systems (ICUAS)*. 2021, pp. 582–590. DOI: 10.1109/ICUAS51884.2021.9476878.

[137] Zhang, Mingyang and Timmerman, Martin and Perneel, Luc and Goedemé, Toon. "Which Is the Best Real-Time Operating System for Drones? Evaluation of the Real-Time Characteristics of NuttX and ChibiOS." In: *2021 International Conference on Unmanned Aircraft Systems (ICUAS)*. Vol. 2021, 582–590. DOI: {10.1109/ICUAS51884.2021.9476878}.

[138] Hommer Zhao. *RTOS vs GPOS: A Complete Guide*. URL: https://www.wellpcb.com/rtos-vs-gpos.html. (accessed: 16.05.2022).