TU WIEN Informatics

Rotating Quadcopter Flight for Collision Avoidance with Static Front Facing LiDAR Sensor

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Technische Informatik

eingereicht von

Alexander Temper, BSc
Matrikelnummer 00928808

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Univ.Prof. Dipl.-Ing. Dr.rer.nat. Radu Grosu
Mitwirkung: Proj.Ass. Dipl.-Ing. Christian Hirsch, BSc

Wien, 1. August 2021

Alexander Temper	Radu Grosu

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

TU WIEN Informatics

Rotating Quadcopter Flight for Collision Avoidance with Static Front Facing LiDAR Sensor

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Computer Engineering

by

Alexander Temper, BSc
Registration Number 00928808

to the Faculty of Informatics

at the TU Wien

Advisor: Univ.Prof. Dipl.-Ing. Dr.rer.nat. Radu Grosu
Assistance: Proj.Ass. Dipl.-Ing. Christian Hirsch, BSc

Vienna, 1st August, 2021

Alexander Temper	Radu Grosu

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Erklärung zur Verfassung der Arbeit

Alexander Temper, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 1. August 2021

Alexander Temper

Danksagung

Ich möchte mich bei Univ.Prof. Dipl.-Ing. Dr.rer.nat. **Radu Grosu**, Leiter der Gruppe Cyber-Physical Systems an der Technischen Universität Wien bedanken, dass ich an diesem Projekt arbeiten konnte. Ich möchte mich auch bei Proj.Ass. Dipl.-Ing. **Christian Hirsch** bedanken, der mich während der Entwicklung und des Verfassens der Arbeit immer auf das Wesentliche fokussiert hielt.

Ein großes Dankeschön geht auch an meine Eltern für ihre Geduld und Unterstützung während meines Studiums in Wien.

Acknowledgements

I want to thank Univ.Prof. Dipl.-Ing. Dr.rer.nat. **Radu Grosu** head of the Cyber-Physical Systems Group at TU Wien for making it possible for me to work on this thesis. I also want to thank Proj.Ass. Dipl.-Ing. **Christian Hirsch** for keeping me focused during the development and feedback on writing the thesis.

Also a big thanks goes to my parents for their patience and support during my study in Vienna.

Kurzfassung

Autonomes Fliegen erfordert immer auch Kollisionsvermeidung, um Schäden am Quadcopter und seiner Umgebung zu vermeiden. In dieser Arbeit wird ein Flugregler auf Basis des *BMF055* entwickelt, der den grundlegenden Flugbetrieb des Quadcopters übernimmt. Für zukünftige Arbeiten und besseres Debugging unterstützt der Flugregler auch *software in the loop* und kann mit einer Simulation (wie *gazebo*) verbunden werden.

Die Kollisionsvermeidung basierend auf dem Rotationsflug und einem nach vorne gerichteten Lidarsensor welcher auf einem separaten Controller entwickelt wurde, um die Einschränkungen des *BMF055* zu umgehen. Dieser dezentrale Ansatz ermöglicht es zukünftigen Arbeiten, den *BMF055* als Ganzes zu ersetzen. Weiters wird ein Test-Quadcopter entwickelt, um die vorgeschlagene Kollisionsvermeidung in der realen Welt zu bewerten.

Die Tests in der Simulationsumgebung und am Prototyp des Quadcopters zeigten, dass aufgrund der Begrenzung der Rotationsgeschwindigkeit des Quadcopters und der Abtastrate von nur einem Sensor eine sichere Kollisionsvermeidung nicht möglich ist. Mit einem Sensor in jede Richtung und einer Rotationsgeschwindigkeit von $180°/s$ konnte der Quadrocopter die meisten Kollisionen vermeiden, aber nicht zuverlässig genug für ein Kollisionsvermeidungssystem.

Abstract

Autonomous flight always requires also collision avoidance to avoid damage of the quadcopter and the environment it operates. In this work a flight controller based on the *BMF055* is developed handling the basic flight operation of the quadcopter. For future work and better debugging the flight controller also supports *software in the loop* and can be connected to a simulation (like *gazebo*).

The collision avoidance based on the rotational flight and a front facing lidar sensor is developed on a separate controller to bypass the limitations of the *BMF055*. This Decentralized Approach enables future works to replace the *BMF055* as whole. A test quadcopter is also developed to evaluate the proposed collision avoidance in the real world.

The tests in the simulation environment and on the prototype quadcopter showed that one front facing sensor is not enough for a save collision avoidance due to the limitation in rotational speed of the quadcopter and the sampling rate of one front facing sensor. With a sensor in each direction and a rotation speed of $180°/s$ the quadcopter was able to avoid most collisions but not reliable enough for a collision avoidance system.

Contents

CHAPTER 1

Introduction

1.1 Motivation

As Drath describes in his work [Dra18] industry 4.0 (4th industrial revolution) is the embedding of internet technology (Internet of Things (IoT)) in the industrial processes. One part of the industry is agriculture, where the IoT is used to collect data from the fields. With this data it is possible to determine the health of the plants or calculate the yield of the field. This kind of agriculture is often called smart farming. One concrete example is a vineyard where the sensors are collecting the humidity and sun exposure on different locations. A challenge in smart farming is the transmission from the sensors to higher abstraction layer where all data can be processed. Often the sensors are powered by a battery and the transmission is done by wireless connections to minimize the installation expenditure. The fields can extend for several kilometres, therefore, these wireless techniques have high requirements in range. But range means more power consumption. Wireless standards which can handle these kinds of use cases (LoRaWAN, Sigfox, LTE-M) are available but the problem is the network coverage which will take a while to build as Schulz describes in [Sch18]. Another approach for collecting data from the sensors is with mobile collectors. These collectors navigate through the field and collect the data from the sensor. A quadcopter suits well for this task because it is capable of breaching a wide range in a short time and can also operate in difficult terrain where ground vehicles have problems. In a productive environment the quadcopter will operate autonomous and therefore, collision avoidance is necessary.

1.2 Aim of this work

The flight controller is the main hardware in a quadcopter comparable with the motherboard of a Personal Computer (PC). It consists of a microcontroller unit (MCU) and interfaces for communication to other important parts of the quadcopter.

In this work we will develop the firmware (operating system) for a quadcopter handling the base flight operation and the collision avoidance. The *BMF055* was predetermined for this work to be used as MCU by the flight controller. The *BMF055* is a System in Package (SiP) which includes three sensors and a MCU. The firmware includes the possibility to be emulated on a PC. The advantage of this is that the hardware of the quadcopter is not needed to run the firmware, which is called Software In the Loop (SITL). To make development and later work easier the quadcopter is put in a simulation environment using the emulation of the firmware. To demonstrate the functionality of the firmware a prototype quadcopter is developed.

1.3 Methodological Approach

First we have to get familiar with quadcopter flight. For this a prototype quadcopter is built with already existing parts (software and hardware). After that the firmware will be developed. This is done by first comparing existing firmwares for quadcopters, and test them on compatibility with the BMF055. A decision is then made for either port an existing firmware to the BMF055 or write an own firmware from scratch. If the base functions of a flight controller are implemented in the firmware the first flight tests and simulations are done. The collision avoidance is then implemented as an additional feature based on the developed flight controller. The collision avoidance will also be tested in simulation and with the developed prototype of the quadcopter.

1.4 Structure of This Work

After the introduction an overview of state-of-the art flight controller and collision avoidance techniques are given. Chapter three discusses two different system design approaches and the proposed collision avoidance mechanism. In chapter four the hardware of the prototype, in its different development states, is shown. In chapter five the software of the prototype and the firmware is presented. In chapter six a simulation of the quadcopter is introduced and the tests with the prototype are discussed. In chapter seven we will summarize what was achieved and how future projects can benefit from the work done.

CHAPTER 2

State-of-the Art

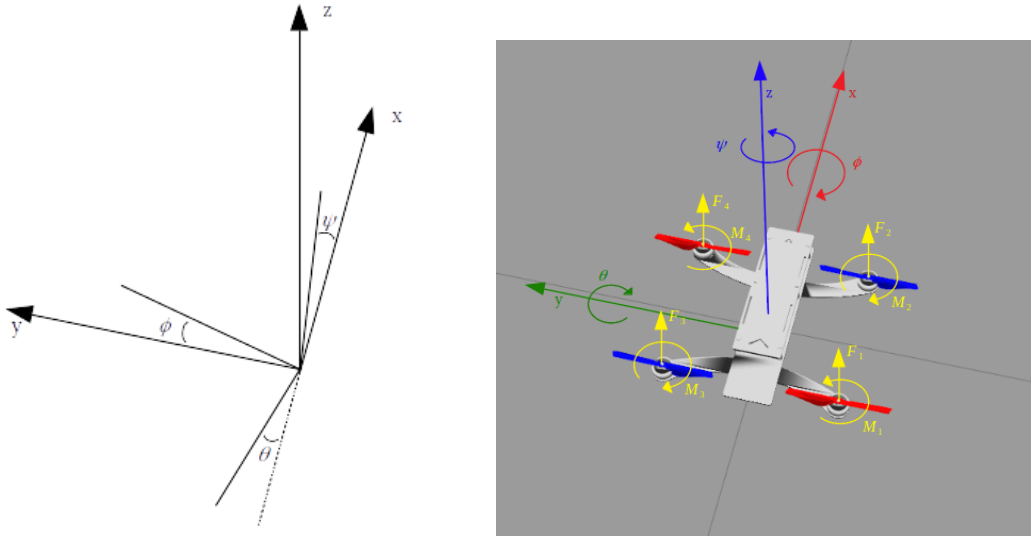
In this chapter we will first discuss the flight dynamics and basic terminology of a quadcopter. After that we will take a look how the quadcopter can estimate its attitude and how the basic controlling for a quadcopter works. We will then present communication protocols used in this work. Before we will discuss the state-of-the art flight controller firmwares we will give an overview about different flight modes implemented by these firmwares. Last but not least we will briefly present collision avoidance techniques.

2.1 Quadcopter Dynamics and Definitions

For controlling and simulating the quadcopter, definition of the quadcopter dynamic model is needed [Luu11][LN16]. First we define two orthonormal coordinate frames to describe the orientation of the quadcopter [SAI18] (see Figure 2.1). The inertial frame or fixed-frame is defined as an unmoving reference the quadcopter is put in [CHR]. ξ is the absolute position of the quadcopter in the inertial frame. The angular position (attitude) η can be defined with three Euler angles ϕ as roll, θ as pitch and ψ as yaw. $\mathbf{V}_i$ defines the linear velocity.

$$\xi = \begin{bmatrix} x \\ y \\ z \end{bmatrix} \quad , \quad \eta = \begin{bmatrix} \phi \\ \theta \\ \psi \end{bmatrix} \quad , \quad \mathbf{V}_i = \begin{bmatrix} v_{x,i} \\ v_{y,i} \\ v_{z,i} \end{bmatrix} \tag{2.1}$$

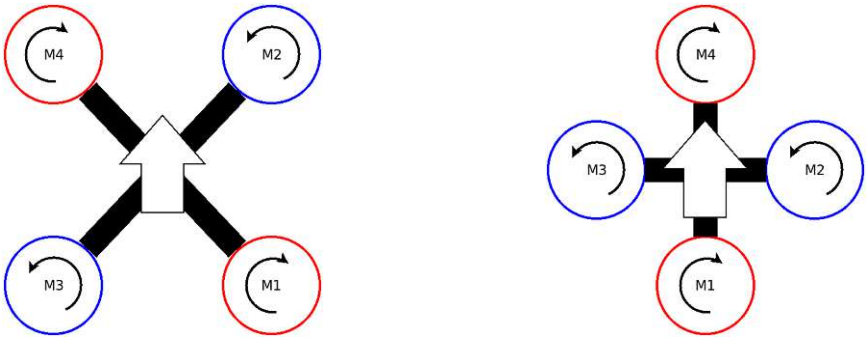
The second coordinate frame is the body frame. This frame has its origin in the center of mass of the quadcopter and is fixed for the body of the quadcopter.



(a) (b)

Figure 2.1: Coordinate frames of the quadcopter. In (a) the inertial frame is shown and in (b) the body frame is shown.

The developed quadcopter prototype also has some characteristics which makes the dynamics easier [LN16]. The quadcopter is symmetrical in geometry, mass and propulsion system. Also the quadcopter is a rigid body, this means it cannot be deformed. To ensure these characteristics there are two arrangements commonly used (see Figure 2.2). Both arrangements make sure that all motors are of the same type with the same propeller and equally far away from the center of mass. The flight controller is mounted in the origin of the body frame. We decided to use the quadcopter in *X-Quad* configuration.



(a) (b)

Figure 2.2: Quadcopter motor arrangements: (a) X-Quad , (b) + Quad

.

In *X-Quad* configuration the quadcopter has four rotors and each rotor is generating a thrust force T_i in the z-axes of the body frame [LN16]:

$$T_i = C_T \omega_i^2 \tag{2.2}$$

C_T is the motor thrust constant (also motor constant) and ω_i the rotational speed in revolution per minute for the i-th motor.

The combined thrust T produced by the four motors is now:

$$T = \sum_{i=1}^{4} T_i = C_T \sum_{i=1}^{4} \omega_i^2 \quad , \quad \mathbf{T}_B = \begin{bmatrix} 0 \\ 0 \\ T \end{bmatrix} \tag{2.3}$$

This total thrust T points towards the body frame z_B axis (T_B). To transform this force to the inertial frame we need the help of rotation matrices.

The elemental rotation matrices are defined as rotation about the x axis with angle ϕ by:

$$R_x(\phi) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\phi) & -\sin(\phi) \\ 0 & \sin(\phi) & \cos(\phi) \end{bmatrix} \tag{2.4}$$

around the y axis with angle θ:

$$R_y(\theta) = \begin{bmatrix} \cos(\theta) & 0 & \sin(\theta) \\ 0 & 1 & 0 \\ -\sin(\theta) & 0 & \cos(\theta) \end{bmatrix} \tag{2.5}$$

around the z axis with angle ψ

$$R_z(\psi) = \begin{bmatrix} \cos(\psi) & -\sin(\psi) & 0 \\ \sin(\psi) & \cos(\psi) & 0 \\ 0 & 0 & 1 \end{bmatrix}. \tag{2.6}$$

To determine the sign of the angle the right hand rule can be used. Place your right hand on the axis so that your thumb is looking in the same direction as the axis, so the rest of the fingers show the direction of rotation.

With this elemental rotation matrices we can define a rotation matrix $\mathbf{R}_b^i$ which does all three rotation at once and therefore transforms a vector from the body frame to the inertial frame.

$$\begin{aligned} \mathbf{R}_b^i &= R_z(\psi) R_y(\theta) R_x(\phi) \\ &= \begin{bmatrix} \cos(\psi) & -\sin(\psi) & 0 \\ \sin(\psi) & \cos(\psi) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos(\theta) & 0 & \sin(\theta) \\ 0 & 1 & 0 \\ -\sin(\theta) & 0 & \cos(\theta) \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\phi) & -\sin(\phi) \\ 0 & \sin(\phi) & \cos(\phi) \end{bmatrix} \\ &= \begin{bmatrix} \cos(\theta)\cos(\psi) & \sin(\phi)\sin(\theta)\cos(\psi) - \cos(\phi)\sin(\psi) & \cos(\phi)\sin(\theta)\cos(\psi) + \sin(\phi)\sin(\psi) \\ \cos(\theta)\sin(\psi) & \sin(\phi)\sin(\theta)\sin(\psi) + \cos(\phi)\cos(\psi) & \cos(\phi)\sin(\theta)\sin(\psi) - \sin(\phi)\cos(\psi) \\ -\sin(\theta) & \sin(\phi)\cos(\theta) & \cos(\phi)\cos(\theta) \end{bmatrix} \end{aligned}$$

Because a rotation matrix is orthogonal the inverse transformation (from inertial to body frame) can be defined as:

$$\mathbf{R}_i^b = (\mathbf{R}_b^i)^{-1} = (\mathbf{R}_b^i)^T \tag{2.7}$$

With Equation 2.8 we can now transform the thrust to the inertial frame.

$$\mathbf{F}_t = \mathbf{R}_b^i * \mathbf{T}_B \tag{2.8}$$

The forces on the quadcopter are now the earth gravity (-z direction of the inertial frame) and the rotated thrust force $\mathbf{F}_t$.

$$\mathbf{F} = \begin{bmatrix} 0 \\ 0 \\ -mg \end{bmatrix} + \mathbf{F}_t = \begin{bmatrix} 0 \\ 0 \\ -mg \end{bmatrix} + \mathbf{R}_b^i * \begin{bmatrix} 0 \\ 0 \\ T \end{bmatrix} \tag{2.9}$$

From Newton's law we know that

$$F = m * a \tag{2.10}$$

From Equation 2.9 and 2.10 we can define the linear motion of the quadcopter as

$$\dot{\mathbf{V}}_i = \begin{bmatrix} 0 \\ 0 \\ -g \end{bmatrix} + \frac{1}{m} * \mathbf{R}_b^i * \begin{bmatrix} 0 \\ 0 \\ T \end{bmatrix}. \tag{2.11}$$

With the linear motion the quadcopter will also encounter an angular acceleration. We can write according to [FBAS16] the Euler's equation for rigid body dynamics:

$$\boldsymbol{\tau} = \mathbf{J} * \dot{\boldsymbol{\omega}} + \boldsymbol{\omega} \times \mathbf{J} * \boldsymbol{\omega} \tag{2.12}$$

where $\mathbf{J}$ is the inertia matrix and $\boldsymbol{\omega}$ the angular velocities and $\boldsymbol{\tau}$ is the momentum.

The rotational velocities around the axis of the body frame is defined as $\boldsymbol{\omega}_B$

$$\boldsymbol{\omega}_B = \begin{bmatrix} p \\ q \\ r \end{bmatrix}. \tag{2.13}$$

With Equation 2.12 and 2.13 we get:

$$\dot{\boldsymbol{\omega}}_B = \mathbf{J}^{-1} \left(\begin{bmatrix} \tau_x \\ \tau_y \\ \tau_z \end{bmatrix} - \begin{bmatrix} p \\ q \\ r \end{bmatrix} \times \mathbf{J} * \begin{bmatrix} p \\ q \\ r \end{bmatrix} \right) \tag{2.14}$$

Note the Equation 2.14 is defined in the body frame of the quadcopter. If the angular velocities in the inertial frame is needed the formular from [SAI18] can be used:

$$\begin{bmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix} = \begin{bmatrix} 1 & \sin(\phi)\tan(\theta) & \cos(\phi)\tan(\theta) \\ 0 & \cos(\phi) & -\sin(\phi) \\ 0 & \sin(\phi)/\tan(\theta) & \cos(\phi)/\cos(\theta) \end{bmatrix} * \begin{bmatrix} p \\ q \\ r \end{bmatrix} \quad for \quad \theta \neq \frac{\pi}{2} \tag{2.15}$$

The momentum $\boldsymbol{\tau}$ used in Equation 2.12 is a reaction of the quadcopter to the spinning of the rotors. From Newton's third law we know that a force always produces a contrary force. This means if the rotor is spinning clockwise the quadcopter reacts by spinning counterclockwise.

As Mellinger showed in his work [Mel12] the momentum generated by one rotor is now defined:

$$M_i = C_D \omega_i^2 \tag{2.16}$$

Where C_D is the rotor momentum constant.

Therefore, in *X-Quad* configuration the two diagonal motor spin in the same direction and the two other diagonal motor spin in the opposite direction so introduced momentum on the bodys z axis z_B sums up to zero.

$$\tau_z = \sum_{i=1}^{4} M_i = \begin{bmatrix} 0 \\ 0 \\ C_D(\omega_1^2 - \omega_2^2 - \omega_3^2 + \omega_4^2) \end{bmatrix} \tag{2.17}$$

The momentum τ_x and τ_y is introduced by the thrust forces T_i. Therefore, for $\boldsymbol{\tau}$ we can write

$$\boldsymbol{\tau} = \begin{bmatrix} l\, C_T(-\omega_1^2 - \omega_2^2 + \omega_3^2 + \omega_4^2) \\ l\, C_T(\omega_1^2 - \omega_2^2 + \omega_3^2 - \omega_4^2) \\ C_D(\omega_1^2 - \omega_2^2 - \omega_3^2 + \omega_4^2) \end{bmatrix}. \tag{2.18}$$

Where l is the distance between the center of mass of the quadcopter and the rotor.

For more information on how to determine the constants C_T and C_D see [LN16].

2.2 Quadcopter Attitude-State Estimation

A quadcopter can use the attitude to achieve flight modes with self level (stabilizing) behavior. The estimation of the attitude state is done by the Inertial Measurement Unit (IMU) which typically consists of a gyroscope and an accelerometer. The gyroscope can measure the angular velocity $\boldsymbol{\omega}_B$. In theory the gyroscope alone can be integrated to estimate the attitude as Ignagni describes in his work [Ign90]. But this requires high precision gyroscopes. An accelerometer measures the acceleration on the three orthogonal sensor axis. It can therefore generate an absolute reference or orientation relative to the earth gravitational field. Unfortunately, if the sensor moves we also measure the acceleration introduced by the movement which leads to high level of noise to the sensor value. Therefore, fusion algorithms are used where the accelerometer corrects the error of integrating the gyroscope. Madgwick showed in his work [Mad] the attitude η can be represented with quaternions because Euler angles have singularities, for example $tang(\pi/2)$. Kuipers [Kui99] gives an introduction to quaternions, which are often used in computer graphics to represent the attitude of an object.

With an IMU consisting of gyroscope and an accelerometer the heading of the quadcopter depends heavily on the gyroscope around the yaw axis because the earth gravity vector is towards this axis as long as the quadcopter is leveled. Therefore the accelerometer cannot help to correct this error and the heading will start to drift over time. To overcome this problem an Attitude and Heading Reference System (AHRS) is can be used. This system adds a magnetometer to the IMU which can be used to correct the heading because it will give an absolute orientation relative to the earth magnetic field.

2.3 Quadcopter Controlling

The most common approach for controlling the quadcopter is a cascading of controllers to split the complexity (see Figure 2.3)[LN16][Mel12]. The *Rate-Controller* controls the angular velocity $\boldsymbol{\omega}_B$ of the quadcopter. The *Attitude-Controller* uses the *Rate-Controller* to achieve a certain attitude $\boldsymbol{\eta_s}$ of the quadcopter. The *Position-Controller* then uses the *Attitude-Controller* to reach a certain place $\boldsymbol{\xi}_s$ in the inertial frame. On cascading controllers it is important that inner controllers run at equal or faster speed compared to the outer controllers providing the set points. Depending on the application area of the quadcopter more or less controller stages are used. The controller itself is mostly implemented as Proportional-Integral-Derivative (PID)-controller.

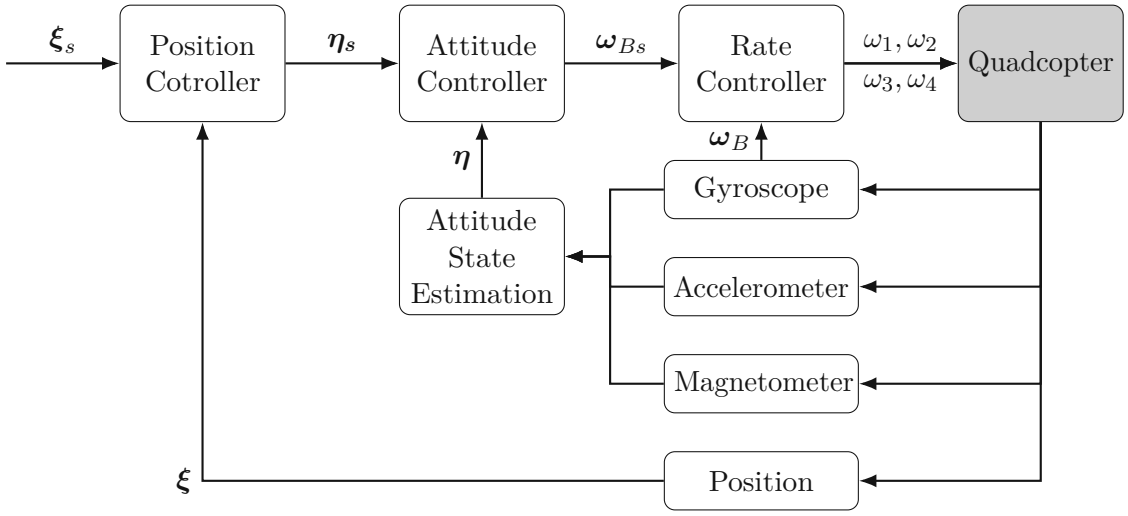


Figure 2.3: Cascading of controller for quadcopter controlling.

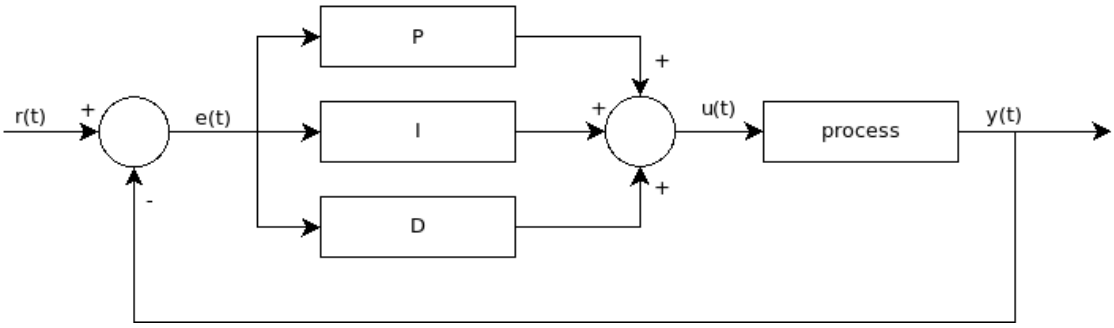


Figure 2.4: PID-Controller block diagram.

2.3.1 PID-Controller

A PID Controller is composed of a proportional, integral and derivative term (see Figure 2.4).

$$e(t) = r(t) - y(t) \tag{2.19}$$

$$u(t) = K_P\, e(t) + K_I \int_0^t e(\tau)\, d\tau + K_D \frac{de(t)}{dt} \tag{2.20}$$

Proportional Term

The proportional term only depends on the current error $e(t)$. Most time you will make K_P value as high as possible to get a fast response on changes of the input. But if the proportional term is too high the controller tends to overshoot and start oscillating also a permanent error is present.

$$u_p(t) = K_P\, e(t) \tag{2.21}$$

Integral Term

The integral term is based on the integration of the error $e(t)$. It can therefore be used to help the proportional term to compensate the permanent error. If the value K_I is too high the controller will also start to oscillate. Often u_t is limited so that the term $u_i(t)$ will not start to windup too much. The result of the wind up is slow response to changes of $r(t)$.

$$u_i(t) = K_I \int_0^t e(\tau)\, d\tau \tag{2.22}$$

Derivative Term

The derivative term is calculated based on the change of $e(t)$ so it can counteract temporal interference and overshooting of the proportional term. If K_D is too high it will counteract the proportional term too much and the controller gets slow.

$$u_d(t) = K_D \frac{de(t)}{dt} \tag{2.23}$$

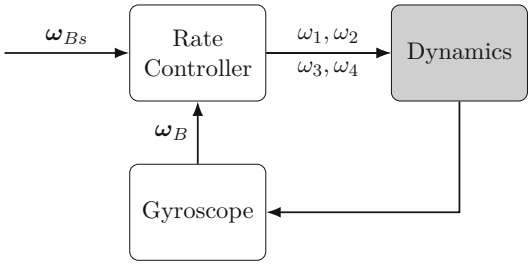


Figure 2.5: Rate Controller diagram.

2.3.2 Rate Controller

The inner loop controller (rate-controller) controls the angular velocity $\boldsymbol{\omega}_B$ of the quadcopter [PP16][LN16][Luu11]. The current angular velocity $\boldsymbol{\omega}_B$ is determined by a gyroscope. The output of the controller are the motors rotational speeds ω_i which leads, according to the quadcopter dynamics, to a change of $\boldsymbol{\omega}_B$. The controller rate differs based on the quadcopter's size. In general, larger quadcopters are more stable so the control rate can be lower.

2.3.3 Attitude Controller

The attitude controller (see Figure 2.6) uses the rate controller to achieve a certain roll and pitch of the quadcopter. Equation 2.1 defines η as the attitude in Euler angles and will be estimated by the *Attitude-State Estimation*. This controller is often implemented as a P-controller generating the setpoint of the angular velocity for the *rate controller*. The controller rate can be slower than the *rate controller* rate because the quadcopter will take some time to response to the setpoint changes of the attitude η_s. Note that yaw is not controlled by the attitude controller and is passed through. This is done because flight modes based on the attitude controller provide the operator the yaw command as an angular velocity which is more intuitive for humans.

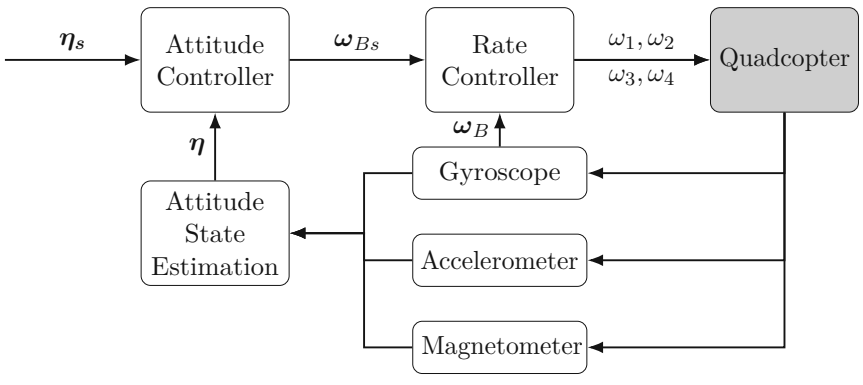


Figure 2.6: Attitude Controller diagram.

2.3.4 Head Free Mode

Because we are planning to let the quadcopter constantly rotate we need a flight mode abstracting the current yaw ψ. This means roll and pitch always stays the same for the operator independent of the current yaw. We use ψ_o to denote the origin yaw angle of the quadcopter. Considered mathematically this is equal to a rotation around the z-axis which can be done either on $\boldsymbol{\eta_s}$ or $\boldsymbol{\eta}$ (see Figure 2.7). We define the current difference between origin yaw angle ψ_o and current yaw angle ψ as

$$\psi_{diff} = \psi - \psi_o \tag{2.24}$$

With Equation 2.24 and 2.6 we get:

$$\begin{aligned} \boldsymbol{\eta}_{s\,r} &= R_z(\psi_{diff})\,\boldsymbol{\eta_s} \\ &= \begin{bmatrix} \cos(\psi_{diff}) & -\sin(\psi_{diff}) & 0 \\ \sin(\psi_{diff}) & \cos(\psi_{diff}) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \phi \\ \theta \\ \psi \end{bmatrix} \\ &= \begin{bmatrix} \phi\cos(\psi_{diff}) - \theta\sin(\psi_{diff}) \\ \phi\sin(\psi_{diff}) + \theta\cos(\psi_{diff}) \\ \psi \end{bmatrix} \end{aligned} \tag{2.25}$$

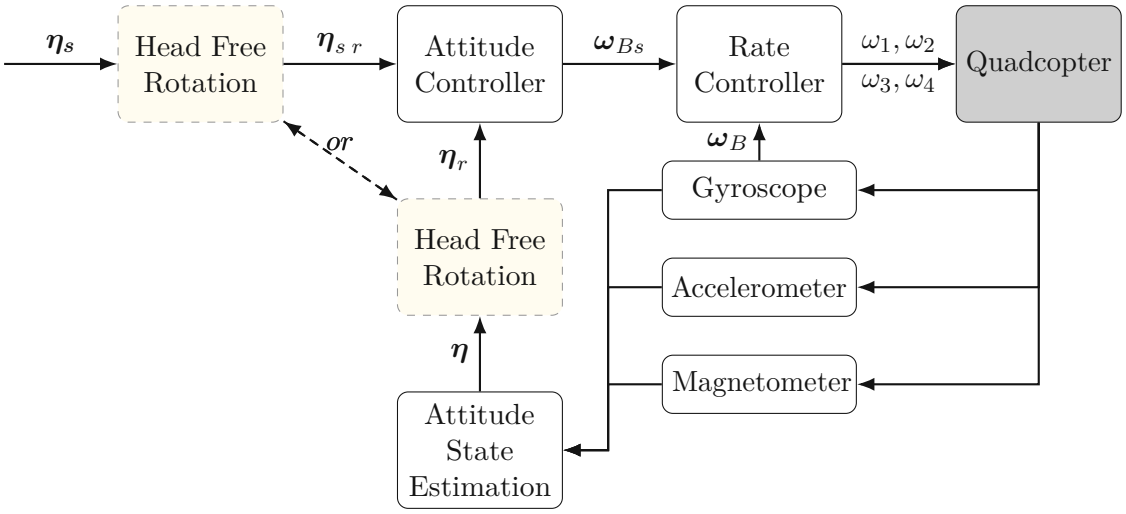


Figure 2.7: Head Free Mode Block diagram.

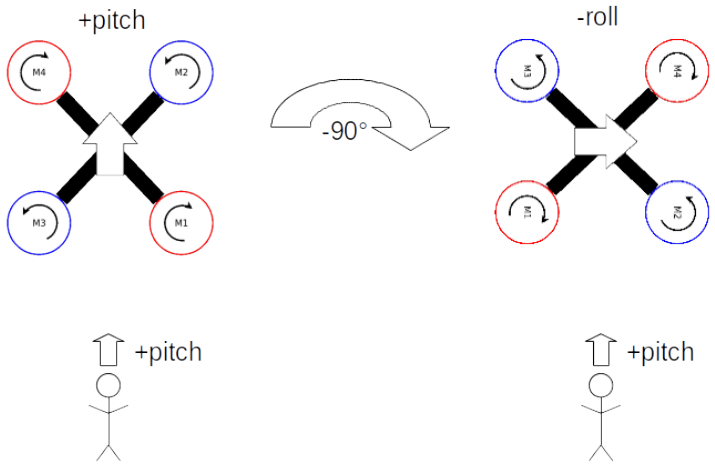


Figure 2.8: Example rotation of -90 degrees around the yaw axis.

Figure 2.8 shows an rotation of -90 degrees around the yaw axis. For the operator pitch stays the same. For the quadcopter we get from Equation 2.25 that now the pitch is transformed to *-roll* see Equation 2.26.

$$\boldsymbol{\eta}_{s\,r} = \begin{bmatrix} \phi\cos(90°) - \theta\sin(90°) \\ \phi\sin(90°) + \theta\cos(90°) \\ \psi \end{bmatrix} = \begin{bmatrix} -\theta \\ \phi \\ \psi \end{bmatrix} \tag{2.26}$$

2.4 Communication Technologies

The quadcopter needs interfaces and protocols to communicate with the sensors on the fields or an operator controlling it. Also communication on the hardware of the quadcopter is necessary, for example the communication between the MCU and Electronic Speed Controller (ESC). Therefore we will introduce some of the used/possible communication technologies.

2.4.1 Bluetooth Low Energy

Bluetooth Low Energy (BLE) is a wireless communication standard introduced in Bluetooth 4.0. Like "classic" Bluetooth it operates on the 2.4 Ghz ISM band. The main focus for BLE is the low energy consumption therefore it suits well for IoT application. This focus on low energy however has the disadvantage that the data throughput is low. For more information see [Joe10].

2.4.2 OneShot125

Quadcopters are often using ESCs to control the rotational speed ω_{1-4} of the motors. Depending on the ESC different protocols can be used. Radio Controlled (RC)-Systems traditionally use Pulse Width Modulation (PWM) with a puls length of $1ms$ to $2ms$ [Baca]. This results in a maximum loop frequency of $500Hz$ for the quadcopter. Modern quadcopters can exceed this bound and are running with $1kHz$ and beyond. Therefore, OneShot125 was introduced as descripted in [Bacb].

As the name suggested the minimum pulse width defined is $125\mu s$ and the maximum pulse width is $250\mu s$. Figure 2.9 shows one period. The decreased pulse width ($2ms$ -> $250\mu s$) enables therefore higher loop frequencies for the quadcopter which results in a faster response of the motor.

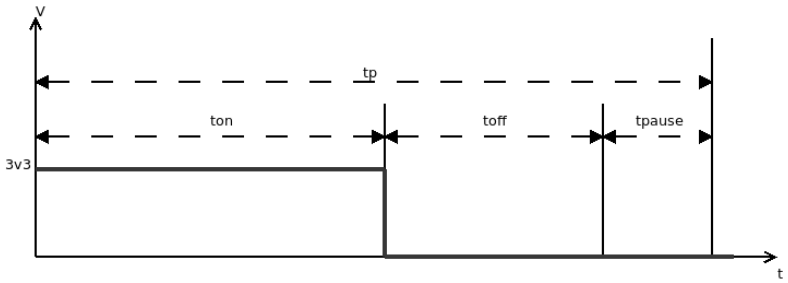


Figure 2.9: OneShot125 timing. $t_p = 300\mu s$, $t_{on} = 125\mu s - 250\mu s$, $t_{off} = 0\mu s - 125\mu s$, $t_{pause} = 50\mu s$

2.4.3 Multiwii Serial Protocol

The Multiwii Serial Protocol (MSP) is a serial request-response protocol used to communicate with the quadcopter. The protocol differs between request, command and response frames shown in Figure 2.10. Each frame is protected by a cyclic redundancy check (CRC) byte. This CRC byte is calculated by a *xor* over the *size*, *type* and *data* bytes of the frame. This simple *xor* for the CRC can lead to unnoticed communication errors. Besides this, *type* and *size* are both 8-bit wide therefore MSP v1 supports 256 different message types with each at most 255 bytes of data. MSP v2 was developed to takle this limitations [Ina].

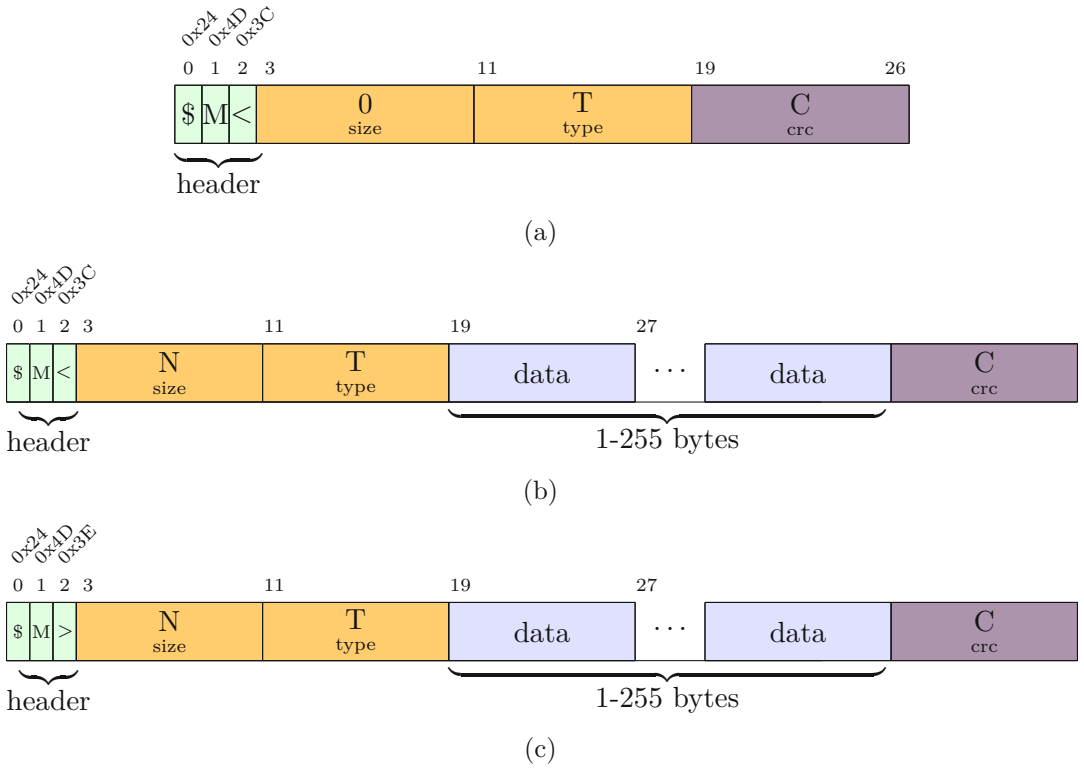


Figure 2.10: MSP frame Types. (a) request frame, (b) a command frame and (c) a response frame.

2.5 Quadcopter Flight Modes

A modern flight controller firmware supports different flight modes. This flight modes differs in the way how the control signal gets interpreted by the quadcopter.

- **Accro Mode**

  In this mode the operator controls the angular velocity of the quadcopter. This mode is used in drone racing because it gives the operator the most direct control of the drone. For this mode only a rate-controller is needed.

- **Angle Mode**

  In this mode the operator controls the quadcopters absolute angle for pitch and roll. This means you can tell the drone for example to pitch +20 degree. This mode is easier to fly because the quadcopter is self stabilizing. This self stabilizing is accomplish by using the attitude information provided by the IMU. In this mode a maximal angle is defined for safety reasons.

- **Horizon Mode**

  This mode is a combination of the *Angle* and *Accro Mode*. Around zero on roll and pitch the quadcopter will start to self stabilize (*Angle Mode* feature). The controls for roll and pitch are the angular velocity like in *Accro Mode*.

- **Headfree Mode**

  In this mode the heading of the quadcopter is fixed for the operator. This means if the operator yaw the drone roll and pitch still stays the same for the operator.

2.6 Flight Controller Firmware

The performance of a flight controller is determinated by the used MCU more specifically the used processor in it. The supported features of the flight controller is mainly determined by the firmware running on this MCU. We will therefore give here an overview of the most used open source flight controller firmware and will discuss the supported MCUs and features. A comparison of the MCUs is shown in Table 2.1.

2.6.1 MultiWii

Many modern flight controller firmwares are based on MultiWii. As the name suggests this flight controller emerged from a hack of the Nintendo Wii Remote Control [EST+18]. It was first implemented on an Arduino Pro Mini Board using the Nintendo Wii Motion Plus or Nunchuk as sensor unit [Mul]. The Arduino Pro Mini Board uses an *ATmega328P* as MCU. Because of the I/O limitations MultiWii was later ported on the Arduino Mega platform which supports up to 54 I/O Pins.

The quadcopter is controlled with a remote control. The remote control acts as a Transmitter (Tx) and sends the steering information over the air to the Receiver (Rx) connected to the flight controller. MultiWii supports Pulse Position Modulation (PPM) and PWM coded signals, as well as Serial Peripheral Interface (SPI) or Universal Asynchronous Receiver Transmitter (UART) based receiving. The supported protocols are *SBUS*, *SUMD* and *SPEKTRUM*. The Rx signal includes at least 8 channels (roll, pitch, yaw, throttle, aux1, aux2, aux3, aux4). These channels will all be normalized to a range from 1000 to 2000 and are called *rcData*. Roll, pitch and yaw get transferred to a range from [-500 to +500] and can be combined with an exponential curve and a deadband. The exponential curve is also applied to the throttle. This is also the place where the rotation of the *rcData* for the *headfree mode* is applied. After that a Throttle PID Attenuation (TPA) is applied. With TPA it is possible to dampen the *rcData* for roll and pitch above a configurable value of throttle to get rid of oscillation on high throttle values. The so manipulated *rcData* is then stored as *rcCommand* which is then used for the PID controller to actually control the quadcopter.

The IMU used by MultiWii is based on a complementary filter and uses the gyroscope and accelerometer. The controller loop is based on a PID controller which implements different modes for controlling the drone. MultiWii supports the accro, angle, horizon and headfree flight mode.

2.6.2 Baseflight

Baseflight [Bas] is a 32 bit fork of MultiWii and written in C. Baseflight is primary developed for the (MCU) *STM32F103CB*. The core functionality of MultiWii stays the same.

2.6.3 Cleanflight

Cleanflight is based on Baseflight and still under development. It shares many parts of the firmware with Betaflight. In the code it is still recognizable that it originated from MultiWii because many features and ideas are still present. The MCU used are all based on the STM32 series of *STMicroelectronics* [Lia]. Its current development is intertwined with Betaflight [Cle] where it is constantly forking from Betaflight (see Figure 2.11). Because of this close relation many tools for Cleanflight are also available for Betaflight and vise versa.

2.6.4 Betaflight

Betaflight focuses on flight performance, leading-edge feature additions and wide target support [Beta]. Many features get merged to Cleanflight if they can prove themselves. The supported MCU are the same as for Cleanflight. For configuration and analyzing Betaflight provides tools.

With the *Betaflight Configurator* [Betc] it is possible to manage the flight controller firmware. This is done by communicating over a serial interface which uses the MSP protocol. The tool is based on JavaScript and is released as Chrome extension. Betaflight also supports a SITL build therefore the *Betaflight Configurator* can also connect to a flight controller via a Transmission Control Protocol (TCP) Socket.

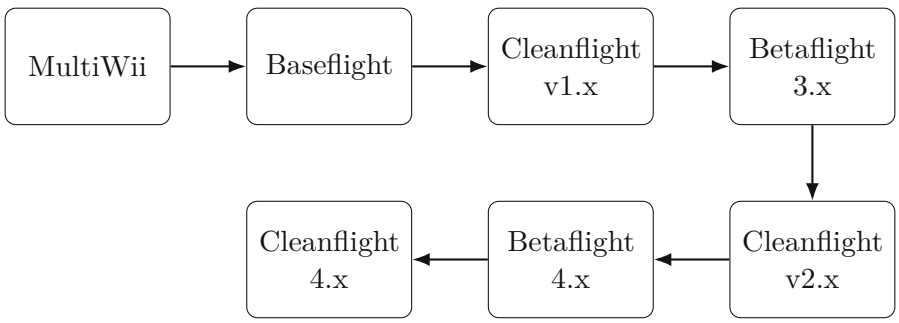


Figure 2.11: Cleanflight fork history

For tuning and debugging the Betaflight firmware the *Betaflight Blackbox Explorer* [Betb] can be used. It is an offline tool in the senses that it uses data logged and stored by the Betaflight firmware. To minimize the space needed for the log, compression methods are use for more information [Bete].

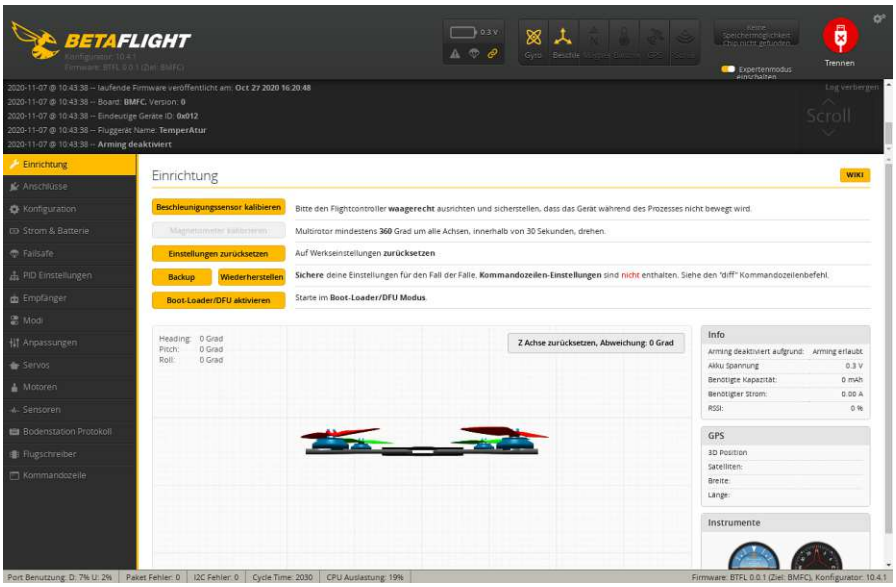


Figure 2.12: Betaflight Configurator

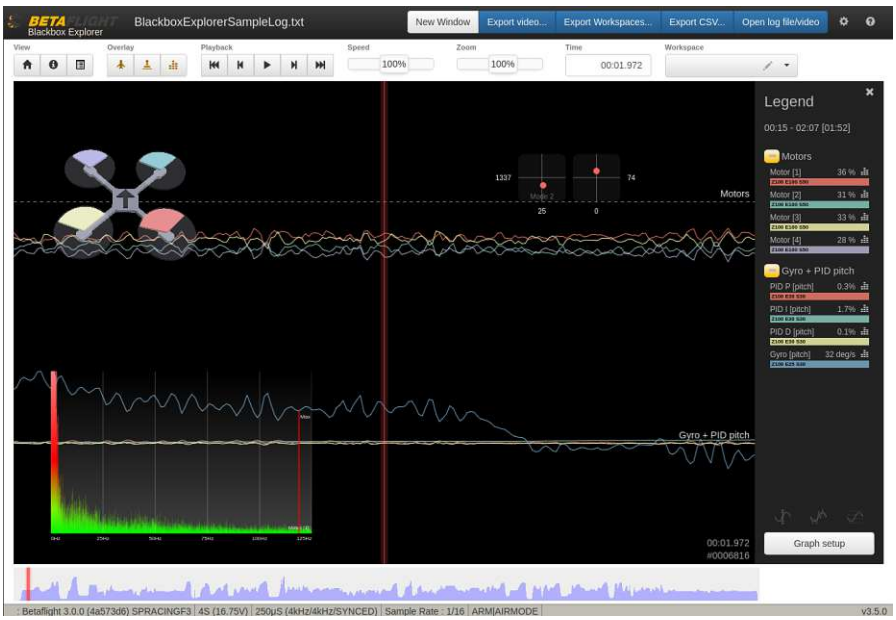


Figure 2.13: Betaflight Blackbox Explorer

2.6.5 NightHawk's BMF055-flight-controller [Blo]

This firmware is inspired by MultiWii 2.1 and takes some improvements done in Cleanflight and Betaflight. The MCU supported is the BMF055. It supports the same flight modes introduced in MultiWii. The sensor fusion algorithm is also based on the MultiWii implementation.

2.6.6 ArduPilot

ArduPilot [Ardb] sets the focus on autopilot operation. This means the quadcopter is capable of operating without human interaction. ArduPilot supports many types of vehicles like copter, plane and rover (see Figure 2.14).

It was first created in 2009 for the *ATmega328* but raptly further developed to a whole ecosystem. This means also the hardware requirements raises. For a complete list of supported hardware see [Arda]. For example the Pixhawk 4[PX4a] uses the *STM32F765* as well as a *STM32F100* as co-processor for I/O tasks. ArduPilot firmware and tools are one of the leading open source autopilot software available and is used a lot in scientific studies and therefore supports an full SITL build.

2.6.7 Crazyflie

Crazyflie is an open source nano-quadcopter development platform from Bitcraze. It is currently available in its second iteration *Crazyflie 2.1*. As MCU the *STM32F405* and for communication the *nRF51822* is used. Crazyflie is often used in scientific studies and supports simulation [SAI18].

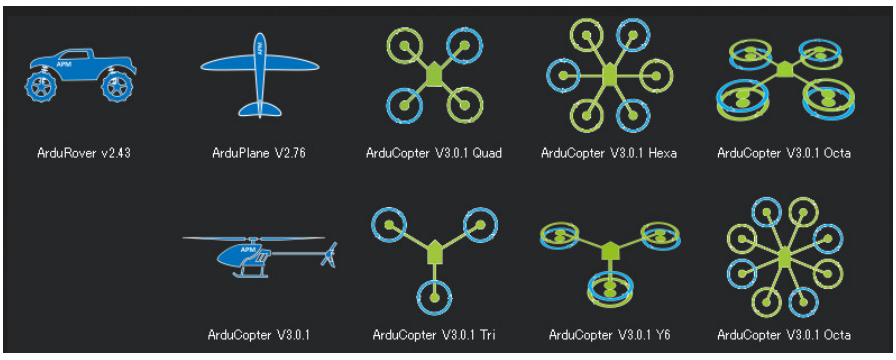


Figure 2.14: Available Firmware variants of ArduPilot.[Ardb]

Table 2.1: Comparison of microcontroller unit of flight controller. [Lia]

MCU	Processor	Ram	Rom	features
ATmega328P	$16MHz$	$2kB$ SRAM	$32kB$ Flash $1kB$ EEPROM	- 8-bit processor
ATmega2560	$16MHz$	$8kB$ SRAM	$256kB$ Flash $4kB$ EEPROM	- 8-bit processor
BMF055	$48MHz$	$32kB$ SRAM	$256kB$ Flash	- 32-bit ARM Cortex M0
STM32F103CB	$72MHz$	$20kB$ SRAM	$128kB$ Flash	- 32-bit ARM Cortex M3 - SCMD - DMA
STM32F303CCT6	$72MHz$	$40kB$ SRAM	$256kB$ Flash	- 32-bit ARM Cortex M4 - SCMD - DMA - FPU
STM32F405RGT6	$168MHz$	$192kB$ SRAM	$1MB$ Flash	- 32-bit ARM Cortex M4 - SCMD - DMA - FPU
STM32F765	$216MHz$	$512kB$ SRAM	$1-2MB$ Flash	- 32-bit ARM Cortex M7 - SCMD - DMA - FPU
STM32H750	$480MHz$	$864kB$ SRAM $128kB$ L1 cache	$138kB$ Flash	- 32-bit ARM Cortex M7 - SCMD - DMA - FPU

- **SCMD** Single-cycle multiplication and hardware division
- **FPU** Floating Point Unit
- **DMA** Direct Memory Access
- **I2C** Inter-Integrated Circuit

2.7 Collission Avoidance

Gageik et al. proposed in their work [GMM12] to split a collision avoidance system into two tasks *obstacle detection* and *collision avoidance*. The obstacle detection is responsible for detecting the obstacle the collision avoidance then tries to avoid the detected obstacles.

For obstacle detection different systems are available. For example Gageik et al. proposed sensors based on ultrasonic. In [MDCDW+17] K. McGuire et al. uses a stereo-camera and image processing. Also Lidar, Time of Flight (TOF) or Infrared sensors are used to sense the environment. If a 360 degree sensing is important multiple sensors are used like in [GMM12] or a rotating sensor as Zheng et al. used in their work[ZZTL19].

The collision avoidance is now responsible to avoid a collision by setting the right action for the quadcopter. The right action depends now on the knowledge the quadcopter has about the environment. If the quadcopter knows the position and has a operating path one right action is to adapt the path to avoid the collision for example like Dentler et al. proposed in there work [DKMV16]. If neither a plan nor the position information is available to the quadcopter the right decision could be to simple limit the steering operation towards the obstacle for example done in [GMM12].

CHAPTER 3

Quadcopter System Design

In this chapter we will discuss two possible system designs of the quadcopter and the proposed collision avoidance mechanism based on the self rotation of the quadcopter.

3.1 System Design

The system is designed around the BMF055 therefore we will first take a closer look to this SiP and then propose two systems designs based on it.

3.1.1 BMF055

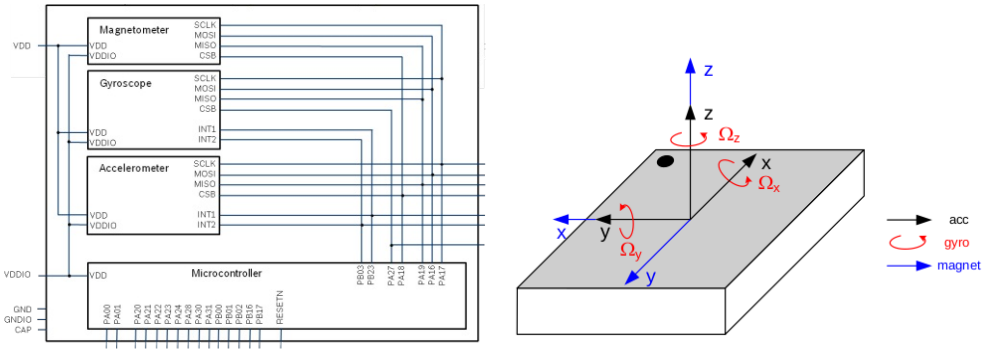
The BMF055 is a SiP which includes three sensors and a MCU (see the datasheet [Bos15] for detailed information). The included sensors are an accelerometer, gyroscope and a magnetometer. Figure 3.1a shows the system architecture of the SiP and Figure 3.1b shows the alignment of the sensors. With these features set the BMF055 is well suited to be used as a quadcopter flight controller.

- **Accelerometer**

  The accelerometer is equivalent to the BMA280. It consists of three axis each with 14 bits of data. A selectable acceleration ranges of $\pm 2g/\pm 4g/\pm 8g/\pm 16g$. The communication is done via SPI with up to $2kHz$ data rate.

- **Gyroscope**

  The gyroscope is equivalent to the BMG160. It consists of three axis each with 16 bits of data. A selectable ranges of $\pm 125°/s, \pm 250°/s, \pm 500°/s, \pm 1000°/s, \pm 2000°/s$. The communication is done via SPI with up to $2kHz$ data rate.



(a) (b)

Figure 3.1: In (a) the BMF055 system architecture is shown, (b) shows the sensor alignment of the BMF055 [Bos15].

- **Magnetometer**

  The magnetometer is equivalent to the BMM150. It consists of three axis each with 13 bits of data. It can measure $\pm 1300 \mu T$ on x and y-axis and $\pm 2500 \mu T$ on z-axis. The communication is done via SPI with up to $30 Hz$ data rate.

- **Controller**

  The controller used in the BMF055 is the *samd20j18a* based on an ARM Cortex-M0+ Core from Atmel. The key features are:

  - 256kB flash
  - 32kB ram
  - up to $48 MHz$
  - up to six Serial Communication Interface
  - watchdog timer
  - up to eight 16-bit timer/counter with PWM
  - 32-bit timer/counter possible by combining two 16-bit

The BMF055 has no range sensor or radio equipment therefore we need extra hardware for this. We propose two approaches where the tasks are different distributed among the hardware.

3.1.2 Centralized Approach

In this approach the BMF055 is the main controller handling all the processing for the quadcopter. This includes the basic flight related task like reading and filtering data from accelerometer, gyroscope, magnetometer. Estimating the attitude based on the sensor data (see Section 2.2), running the real time controller loop of the quadcopter (see Section 2.3) and communicating with a ground station or pilot. It also includes services based on the basic flight operation like the collision avoidance, communication to swarm placed on the field or future work like autonomous flight. A sketch of the interaction to other hardware components is shown in Figure 3.2.

3.1.3 Decentralized Approach

In this approach we split the tasks on different hardware. The basic flight related tasks are still done by the BMF055. All other tasks are done by a separate controller connected to the BMF055. We call this controller *Pilot Controller* because for the BMF055 the controller acts as a pilot of the quadcopter. See Figure 3.3 for a sketch of the interaction between the hardware components.

3.1.4 Comparison

In the *Decentralized Approach* the features are distributed on two hardware components, this means each component has less requirements in terms of computational power. The *Decentralized Approach* also has the advantage that the two firmwares of the controller can be used without the other. This means the firmware running on the BMF055 is a full featured flight controller firmware and can be used without the *Pilot Controller*. On the other side the *Pilot Controller* can operate with other flight controllers as the BMF055. Note this is only possible because the interface between the two components is defined with that clear segregation of duties in mind. A disadvantage of this approach is the bandwidth and time needed for this communication between the two components.

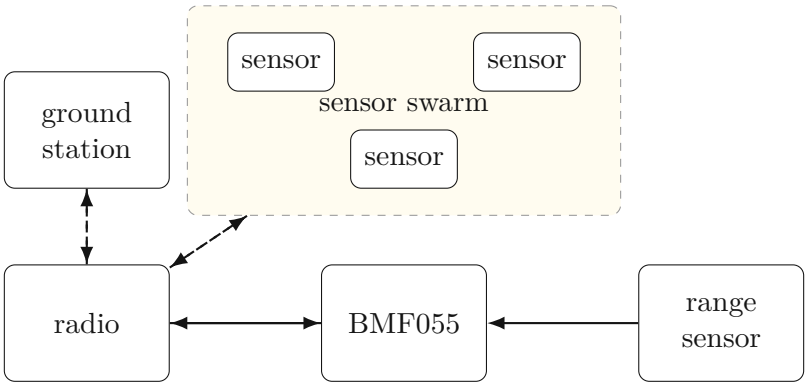


Figure 3.2: Sketch of interaction to the other hardware components for the centralized approach.

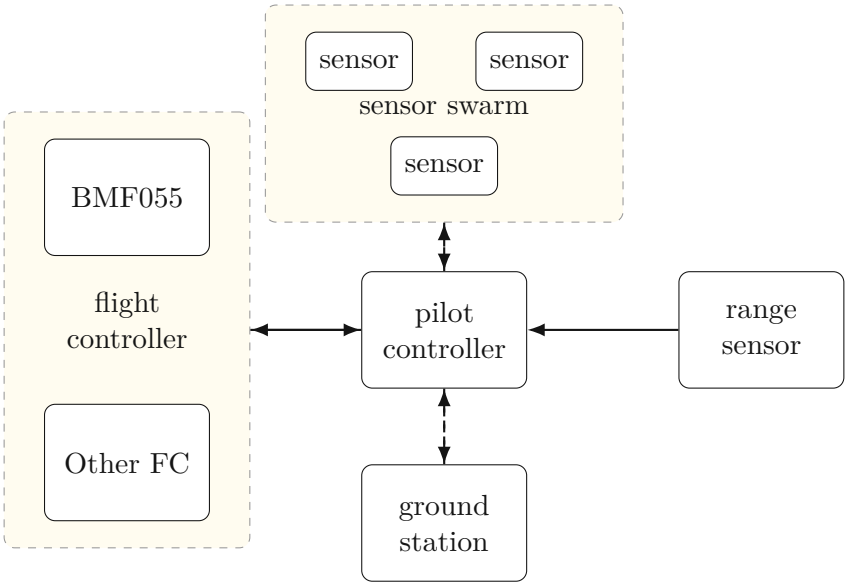


Figure 3.3: Decentralized Approach.

The *Centralized Approach* does not have this kind of bandwidth bottleneck because all information is processed on BMF055 itself but this is also the biggest problem of this approach. The BMF055 is very limited by the *samd20j18a* integrated in it. So implementing all software will be hard because controlling a quadcopter is still a real time task. During early research we already reached limitation of the BMF055 by only implementing the basic flight features. So we decide using a radio equipment implementing the protocol stack for the radio protocol. The used radio equipment includes a MCU for programming. We therefore decide to use the *Decentralized Approach* because we already need an extra MCU by the radio equipment.

3.1.5 Flight Controller Firmware

We give an overview of the State-of-the-Art flight controller firmware in Chapter 2.6. Table 3.1 compares the presented firmwares and their features related to this work. Because the SiP of the quadcopter was predefined we must choose a firmware most suitable for this hardware. From scratch only the NightHawk's BMF055 flight-controller supports the BMF055 as build target but it lacks of compatible to external tool like the *Betaflight Configurator* or *Betaflight - Blackbox Explorer* as well as a hardware abstraction to make a SITL build possible. Therefore, we used the firmware as reference for our first build. This first firmware was an attempt to port Betaflight to the BMF055. But we must admit ourselves during development that this was too ambitious because Betaflight was simply too big in size and computational power to run on the BMF055. Therefore, we decided to build a custom firmware. The firmware partially supports the *Betaflight Configurator* (Version 10.4.1) and *Betaflight - Blackbox Explorer* (Version 3.5.0).

Table 3.1: Comparison of flight controller firmware.

Firmware	BMF055 target	Clock Speed (MHz)	SITL	Simulation	Logger
MultiWii	no	16	no	no	yes
Baseflight	no	72	no	no	yes
Cleanflight	no	168-216	yes	yes	yes
Betaflight	no	168-216	yes	yes	yes
ArduPilot	no	216	yes	yes	yes
Crazyflie	no	168	yes	yes	yes
NightHawk's BMF055	yes	48	no	no	no

3.1.6 Pilot Controller Hardware

The main feature needed for the *Pilot Controller* hardware is to communicate with the sensors placed on the field and with a ground station controlling/observing the quadcopter. During development we used two different micro controllers.

ESP32-WROOM-32

The ESP32 from *Espressif* is a 32-bit micro controller. It is the successor of the ESP8266 often used in home automation. It is programmable via the *Arduino IDE*.

The main features are:

- Dual core Xtensa microprocessor (LX6), running at 160 or 240MHz
- 520KB SRAM
- 4/8/16MB flash
- Wireless interfaces
  - Bluetooth v4.2 BR/EDR and BLE specification
  - 802.11 b/g/n (802.11n up to 150 Mbps)
- Supply voltage $3.3V$

RFD77101 Simblee Module

The *RFD77101 Simblee Module* from RF Digital is a BLE module with a ARM Cortex-M0 processor.

The main features are:

- ARM Cortex-M0 32-bit processor @ 16MHz
- 24kB SRAM
- 128MB flash
- Bluetooth Smart Radio

nRF52840

The nRF52840 from Nordic Semiconductor is an ARM 32 Bit Processor for the IoT uses.

The main features are:

- ARM Cortex-M4F 32-bit processor @ 64MHz
- 256 kB SRAM
- 1MB flash
- Wireless interfaces
  - Near Field Communication (NFC)-A
  - Bluetooth 5 with 2 Mbps, 1 Mbps, 500 kbps, and 125 kbps
  - Thread IEEE 802.15.4-2006: 250 kbps
  - Proprietary 2.4 GHz: 2 Mbps, 1 Mbps
- Wide supply voltage range $+5.5v$ to $1.7v$
- Digital signal processing (DSP) instructions
- Hardware accelerated single-precision Floating Point Unit (FPU) calculations

Comparison

Because the ESP32 is commonly used in home automation it is very easy to program for it via the Arduino IDE. The WLAN support and therefore the high data rates enables good debug capability. The nRF52840 on the other hand is more suitable for our operation area. It supports more wide area low power network types and is also used by the sensors on the field [Tho20]. We therefore use the nRF52840 as *Pilot Controller* hardware. The ESP32 was nevertheless used during development for debugging and logging data during tests.

3.2 Collision Avoidance System

For the collision avoidance we mount a TOF sensor on the quadcopter as shown in Figure 3.4. To enable a 360 degree sensing we let the quadcopter rotate with an angular velocity ω_r. For the operator this rotation is abstracted away by using the head free mode. Because of the flight dynamics this rotation speed has an upper bound ω_{rmax} until the quadcopter can not be controlled anymore. This ω_{rmax} was determined using simulation and the prototype. We encounter that on $\omega_{rmax} = 180°/s$ the quadcopter was still controllable.

The used TOF sensor has a sampling period of $30ms$. With a rotation speed of $180°/s$ and a new sensor value every $30ms$ we get a sensor value every 5.4°. This means we get a sensor value every two seconds for one direction in the inertial frame. With this low sampling rate it is hard for the quadcopter to keep up with changes in the environment. Also the quadcopter does not have a global position in the inertial frame so it does not

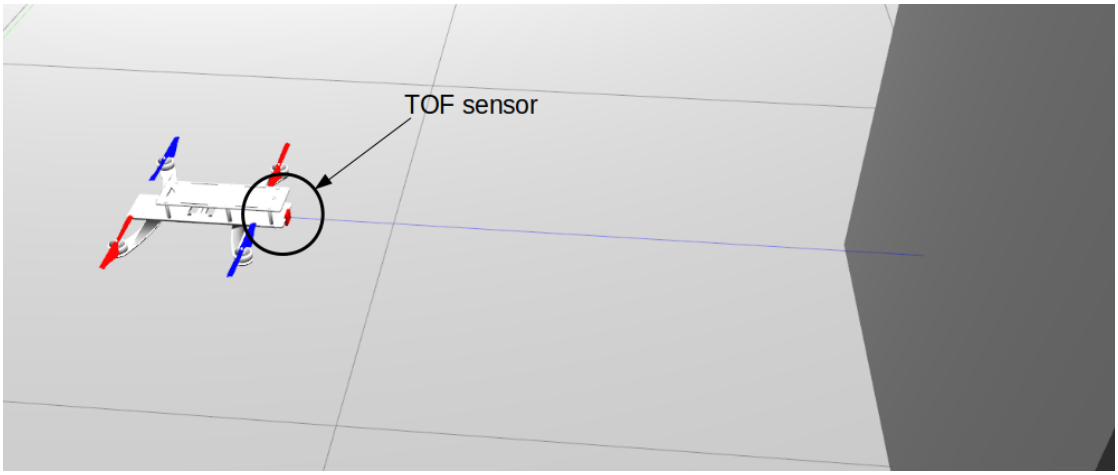


Figure 3.4: Sensor mount of range sensor on the quadcopter.

know where it is. This means it is not possible to build an accurate estimation of the world fast enough to develop a high level collision avoidance based on it.

We decide to implement the collision avoidance based on the current state of the sensor. If the current value of the sensor detects an object the quadcopter reacts with a response to avoid the collision. This response is to mix a appropriate value on pitch and roll to push the quadcopter away from the object or to slow down the approximation to it. In Figure 3.5 the controller diagram of the quadcopter is shown.

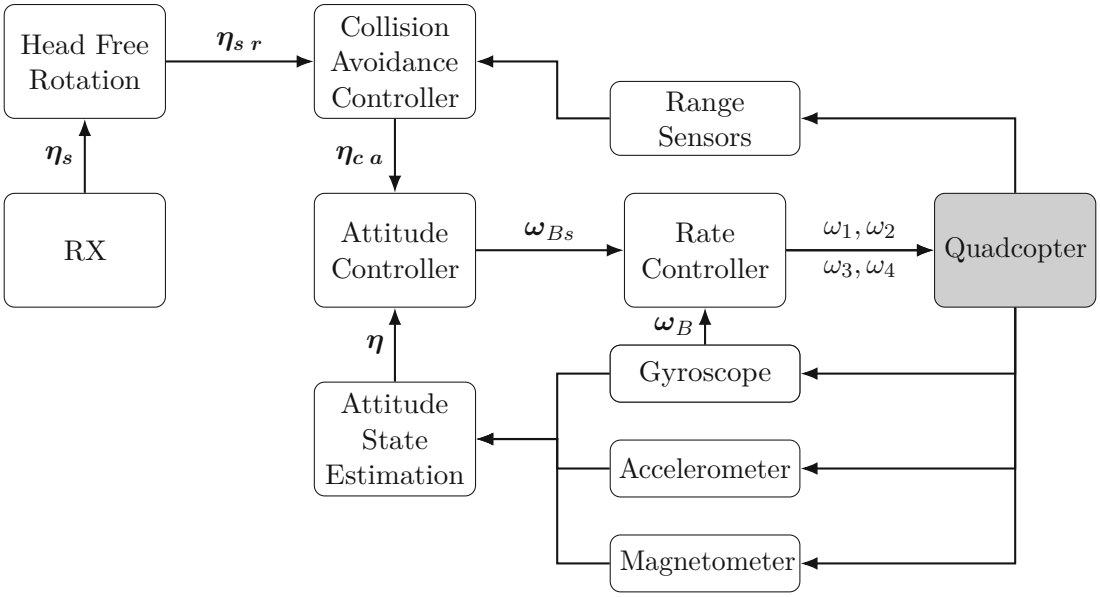


Figure 3.5: Controller structure for collision avoidance.

To improve the low sampling rate we later experimented in the simulation with different sensor arrangements (see Table 3.2).

Table 3.2: Link between sensor arrangement and sampling rate in one direction of inertial frame.

sensor arrangement	sampling rate in one direction of inertial frame
one front facing sensor	2 sec
front facing and back facing sensor	1 sec
sensor in every direction	0.5 sec

The *Collision Avoidance Controller* now is composed of one or multiple *Push-Controller* depending on the sensor arrangement. The *Push-Controller* itself implements a proportion part and a derivative part. Depending on the position of the sensor the *Collision Avoidance Controller* will mix the value differently to roll and pitch in $\boldsymbol{\eta_{s\,r}}$. A diagram of the *Collision Avoidance Controller* is shown in Figure 3.6.

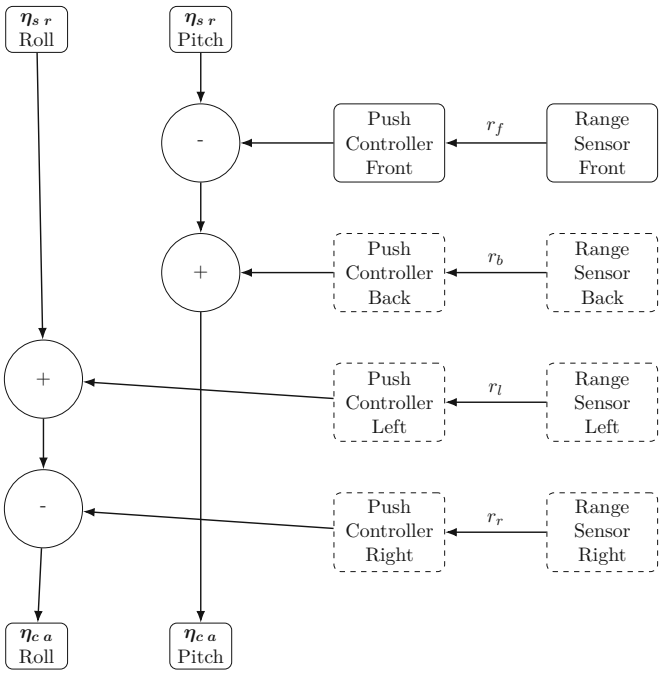


Figure 3.6: *Collision Avoidance Controller* internal structure.

CHAPTER 4

Quadcopter Prototype Hardware

For developing and debugging a prototype quadcopter was build from scratch with components available on the market. In this chapter the parts used are discussed. Several versions of the prototype were developed which mainly differs in the used controllers and feature set.

4.1 Components

The main parts of a quadcopter are frame, motors, propellers, battery, ESC, flight controller and receiver. These parts have to be tuned to each other to make the quadcopter fly. To choose the right parts the weight of the quadcopter is the most important figure. This weight is mainly produced by the frame, motor and the battery.

- **Frame**
  First decision to take was how big the prototype should be. We decide to take the "LHI H280" (see Figure 4.1a). This frame has a diagonal of 280mm between the rotor hubs, is made of carbon and glas fiber, weights 190g and is modular so parts can be changed on damage and we can make adjustments later.
- **Propeller**
  Limited by the frame we can use propellers up to a diagonal of 150mm. We choose the "DAL 5045" (see Figure 4.1b), these propellers are cheap and very robust. They are dual blade propellers with a diagonal of 125mm.

- **Motor**
  The motor in combination with the propeller has to be strong enough to generate enough thrust to lift the quadcopter. We choose the "Hobbywing XRotor-2205-2300KV"(see Figure 4.1c) motors. The manufacture provides a table (see Table 4.1) with the data needed to estimate the thrust one motor can generate. This depends on the voltage used to drive the motor and the propeller the motor is spinning.

- **Receiver**
  The receiver has to be compatible with the chosen flight controller. We use the *FrSky XM PLUS Mini* which uses the SBUS protocol to communicate with the flight controller.

- **Flight Controller**
  As flight controller the "*Matek F405-STD*" was chosen because it is designed around the STM32F4 which is a very popular micro processor for many flight controller firmwares. This flight controller supports Betaflight, INavFlight and ArduPilot.

- **Power Distribution Board**
  We use the *FCHUB-6S* as Power distribution board (PDB). With a PDB we can later easy switch the flight controller without re-wiring the ESC. The *FCHUB-6S* can also be connected to the *Matek F405-STD* with a flat ribbon cable. This board also generates an 5V and 10V voltage supply.

- **Electronic Speed Controllers**
  The motors are driven by the ESC. This controller gets a control signal from the flight controller and regulates the motor [ssk]. The prototype uses brushless motors so the ESC has to generate the desired rotation magnet field. From Table 4.1 we get about 30A as the worst case current draw for one motor. We choose the "Hobbywing X-Rotor 30A Micro 2-4S ESC"(see Figure 4.1d) which can manage this worst case current and also good protocol support to communicate with the flight controller.

(a) (b) (c) (d)

Figure 4.1: Parts of the quadcopter prototype. (a) frame [Lig], (b) propeller [DAL], (c) motor [Hobb] and (d) ESC [Hoba].

Table 4.1: RS2205 trust data [Hobb]

voltage	**current** (A)	**thrust** (g)	**power** (W)	**speed** (RPM)
12	1	62	12.00	6400
	3	162	36.00	10080
	5	236	60.00	12070
	7	311	84.00	13730
	9.1	374	109.20	15100
	11	439	132.00	16320
	13	490	156.00	17350
	15.3	548	183.60	18350
	17.3	611	207.60	19210
	20.7	712	248.40	20080
16	1	72	16.00	7220
	3	183	48.00	10790
	5	283	80.00	13030
	7.1	352	113.60	14720
	9.1	426	145.60	16180
	11	497	176.00	17150
	13	560	208.00	18460
	15	628	240.00	19270
	17	692	272.00	20270
	19	754	304.00	21060
	21	812	336.00	21840
	23.3	878	372.80	22590
	25.4	936	406.40	23210
	27.3	997	436.80	23920
	29.9	1024	478.40	24560

- **Battery**
  The battery is the main part in terms of weight for a quadcopter. The main requirements on the battery are the flight time and the ability to provide enough current for the motor of the quadcopter. So if we take strong motors we need also a battery to drive them otherwise we risk a permanent damage of the battery. We choose a Lithium Polymer (LiPo) battery for the prototype. These kinds of batteries are often used in quadcopters because they are able to provide high burst current with low weight. We decide to use a 3S LiPo Battery, 3S means that 3 LiPo Cells are connected in serial. With a voltage of 3.7V to 4.2V per cell we get a voltage of 11.1V to 12.6V.

  Two different batteries were used:

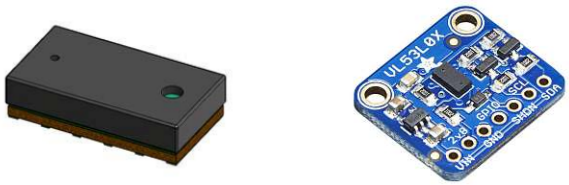
  - Tattu 10010 RC Battery 1550mAh 75C 3S
    - weight **136g**
    - max. continuance current draw **116.25A**
    - max. burst current draw **232.5A**
  - Conrad energy(LiPo) 11.1V 3800mAh 3S 20C
    - weight **326g**
    - max. continuance current draw **76A**
    - max. burst current draw **152A**

Table 4.2: vl530x default range profiles [STM16]

Range profile	Timing budget	Typical max range	typical application
Default mode	30ms	1.2m (white target)	standard
High accuracy	200ms	1.2m (white target)	precise measurement
Long range	33ms	2m (white target)	long ranging, only for dark conditions
High Speed	22ms	1.2m (white target)	high speed where accuracy is not priority

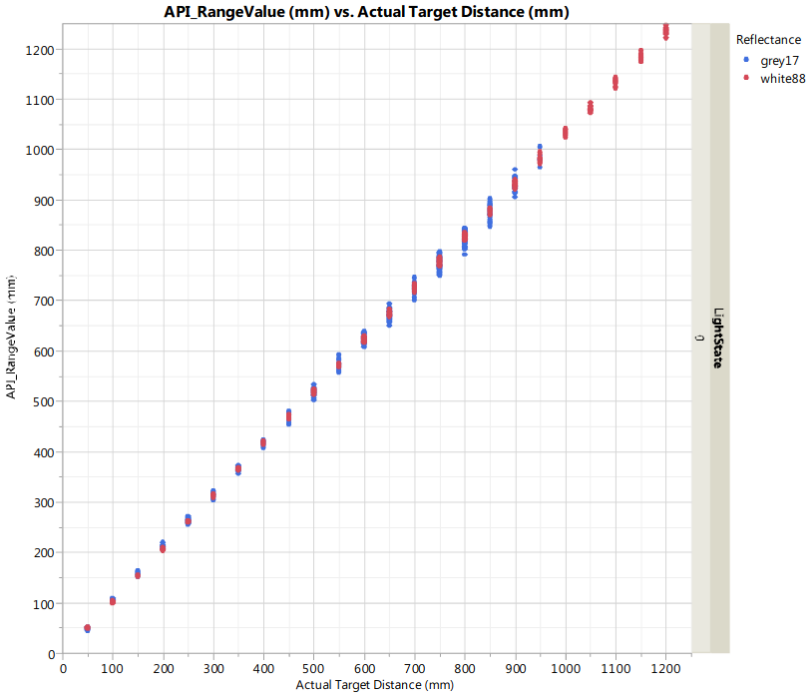
- **Range Sensor**
  As range sonsor we choose the VL53L0X from STMicroelectronics. This TOF sensor is available with a breakout board (Adafruit VL53L0X, see Figure 4.2b). The sensor is able to measure a range up to 2 meters. The reported range is independent of the target reflectance. The VL53L0X provides an Inter-Integrated Circuit (I2C) interface for configuration and getting range data.

  The API (see datasheet [STM18]) provided by STMicroelectronics supports by default four different range profiles see (Table 4.2). It is also possible to define own range profiles. In Figure 4.3a measurements for the default profile and in Figure 4.3b measurements for the long profile are shown.

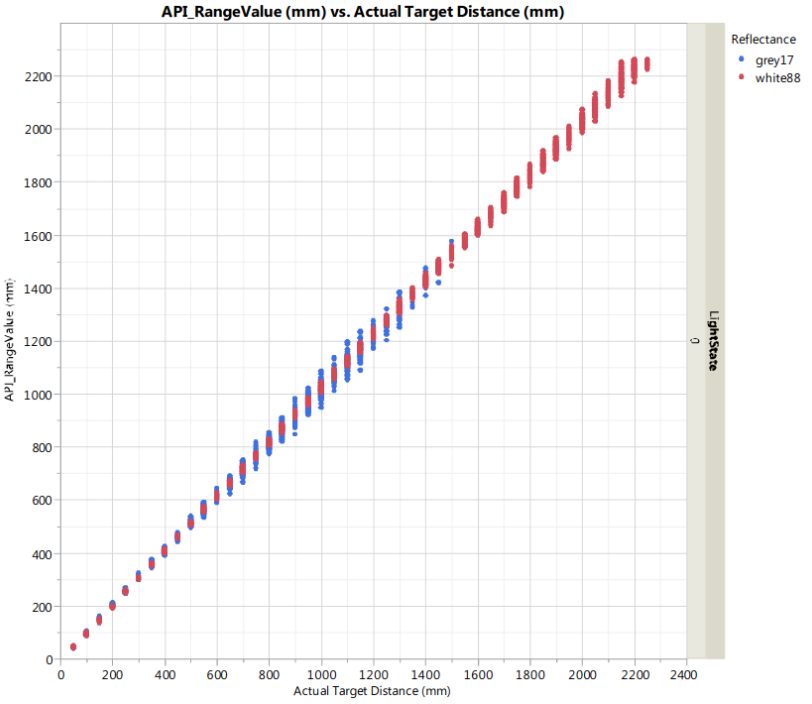


(a) (b)

Figure 4.2: In (a) the package of the VL53L0X is shown [STM18] , (b) shows the breakout board from *Adafruit*.



(a)



(b)

Figure 4.3: Range measurements for the VL53L0X [STM18]. In (a) the results of the default profile in (b) the results of the long profile is shown.

4.2 Initial Build

The first finished build was to get familiar with quadcopters and is sketched in Figure 4.4. The weight of the quadcopter without battery is about 380g (see Table 4.3). The maximum thrust of all four motors is about 2.8kg which is enough even under extreme condition (for example catch quadcopter after free fall).

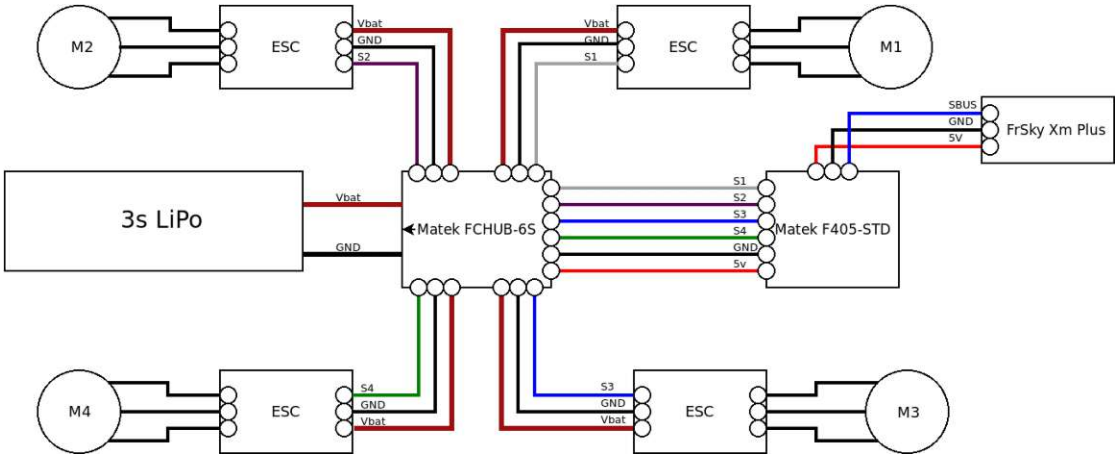


Figure 4.4: Sketch for the first build of the quadcopter prototype.

Table 4.3: Prototype weight

component	**weight** (g)
frame	190
propeller	20
motors	113.6
ESC	24
PDB	8.5
receiver	1.6
sum	357.7
battery	
Tattu	136
sum	**493.7**
battery	
Conrad energy	326
sum	**683.7**

4.3 Altitude Hold Build

After the first manual flights with the quadcopter an altitude hold build was made. In this build the flight controller was not longer controlled over the receiver chosen in 4.1. Instead a controller with BLE was mounted on the quadcopter and acts as a gateway between the flight controller and a PC. On the PC the Robot Operating System (ROS) is used to get data from a joystick and sends them to the gateway. The BLE hardware used on the PC is the LogiLink BT0015.

For the BLE controller an *RFD77101 Simblee Module* was used. There are GIPO breakout boards (see Figure 4.6) available so mounting was no problem. To hold the altitude the quadcopter needs to know how far the ground is away. This was done by a down facing range sensor (VL53L0X). Figure 4.5 shows the complete build.

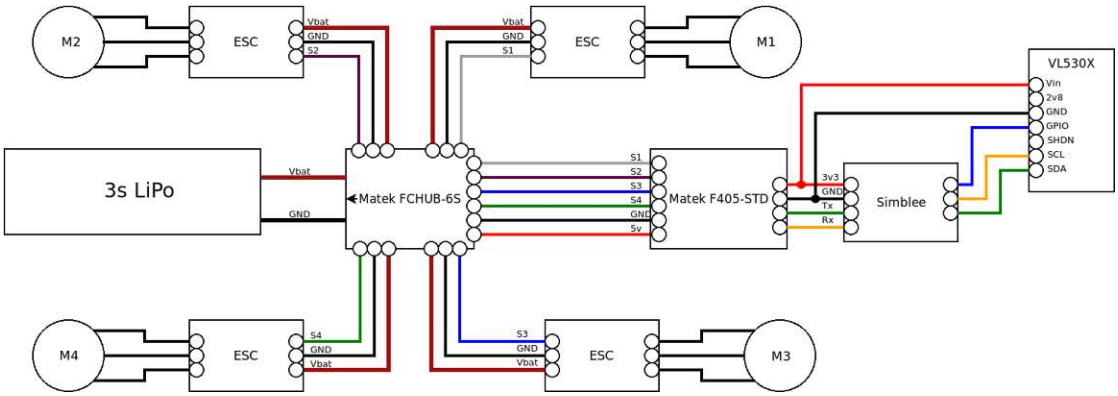


Figure 4.5: Sketch for the *Altitude Hold Build* of the quadcopter.

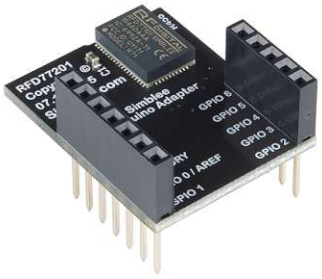


Figure 4.6: *Simblee* breakout board [Spa].

4.4 BMF055 Build

In this build the *Matek F405-STD* flight controller was replaced by the flight controller based on the BMF055. For the BMF055 we used the *BMF055 Shuttle Board* (see Figure 4.8) available from *Bosch*. We also need a voltage regulator (L4931) to generate the 3,3V used by BMF055, *Simblee* and VL530X. The build is shown in Figure 4.7.

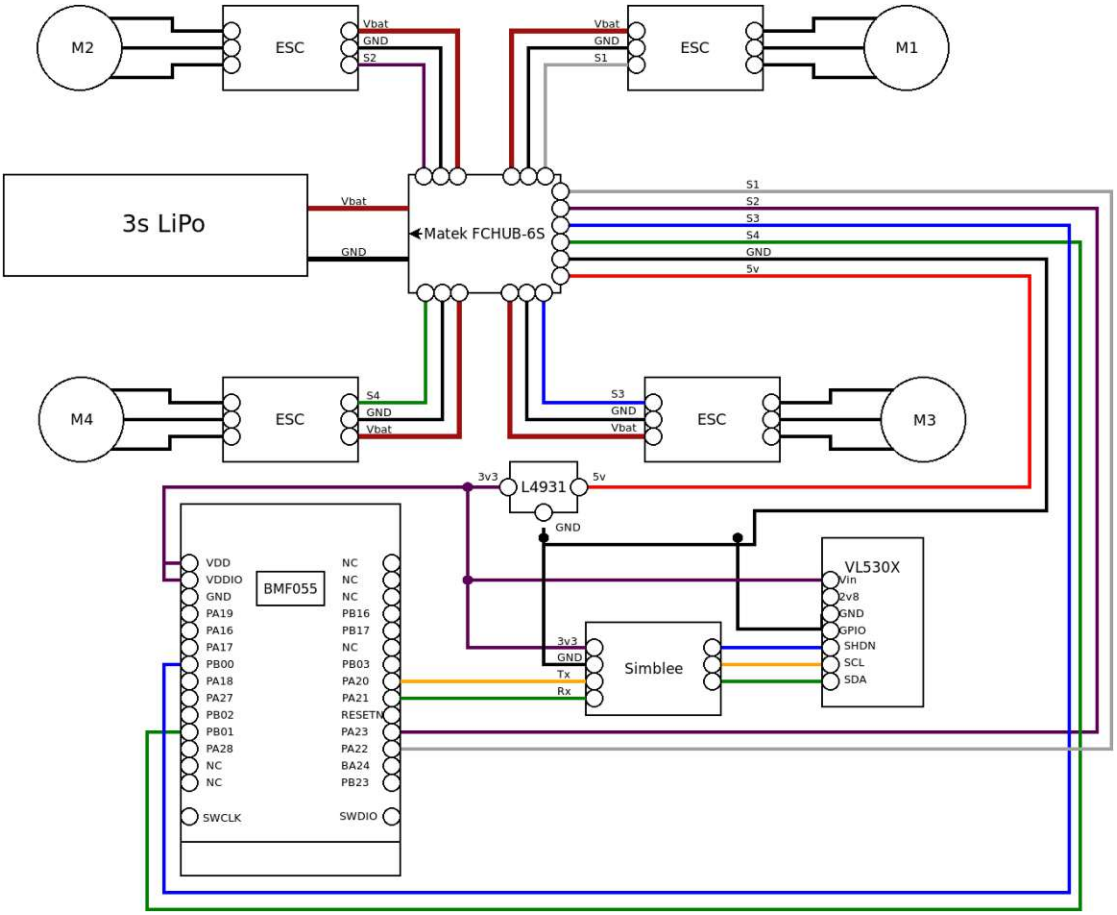


Figure 4.7: Sketch of the quadcopter prototype using the BMF055 as flight controller.

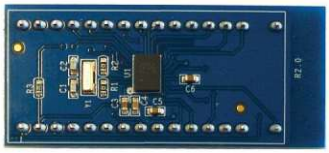


Figure 4.8: BMF055 Shuttle Board [Bos16].

4.5 Final Build

In this build the *Simblee* module was replaced by the *Particle XENON*. We add a front facing range sensor for the collisions avoidance and status light-emitting diode (LED)s for indicating the internal state of the quadcopter to the user (see Figure 4.9).

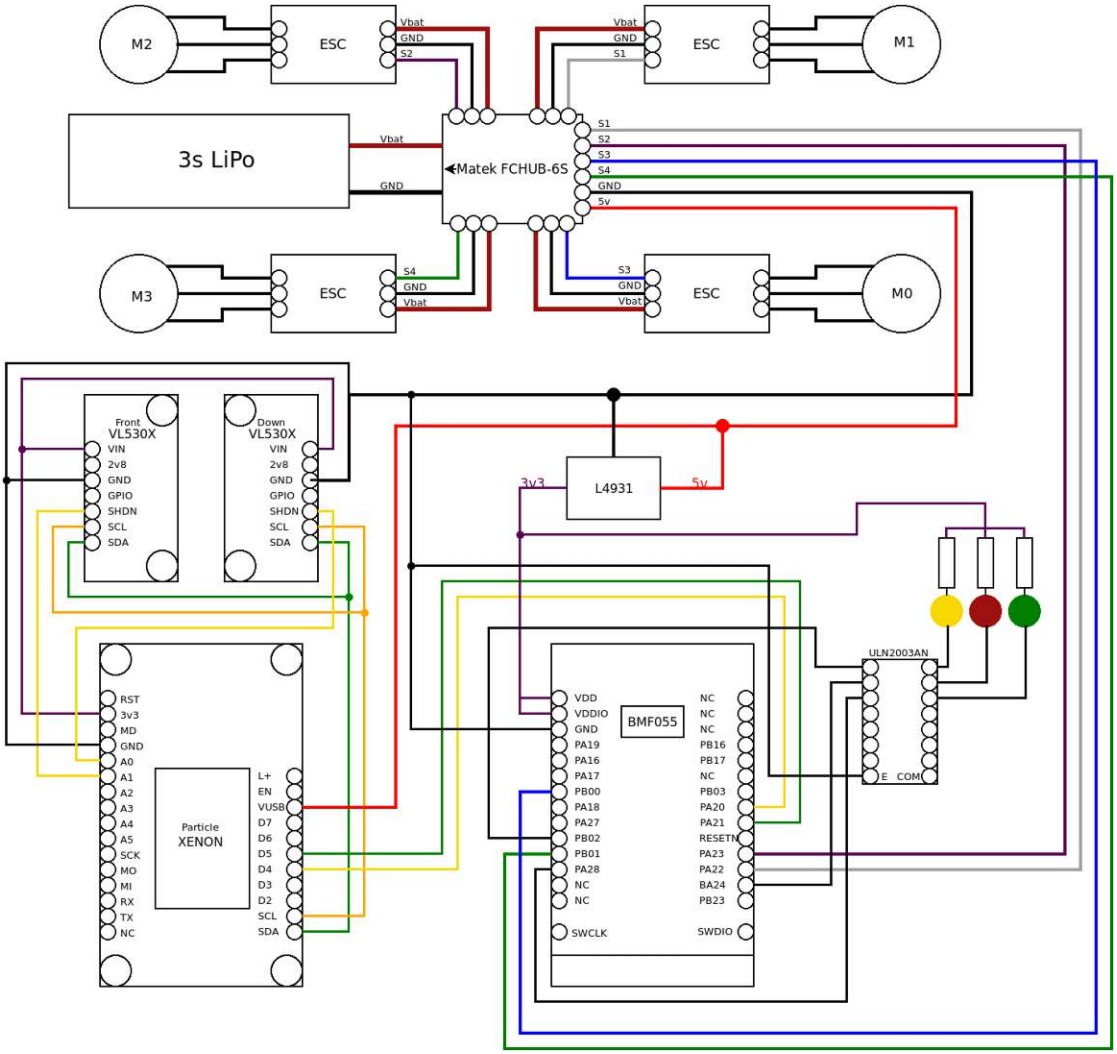


Figure 4.9: Sketch of the quadcopter prototype using the BMF055 as flight controller and *Particle XENON* as pilot controller.

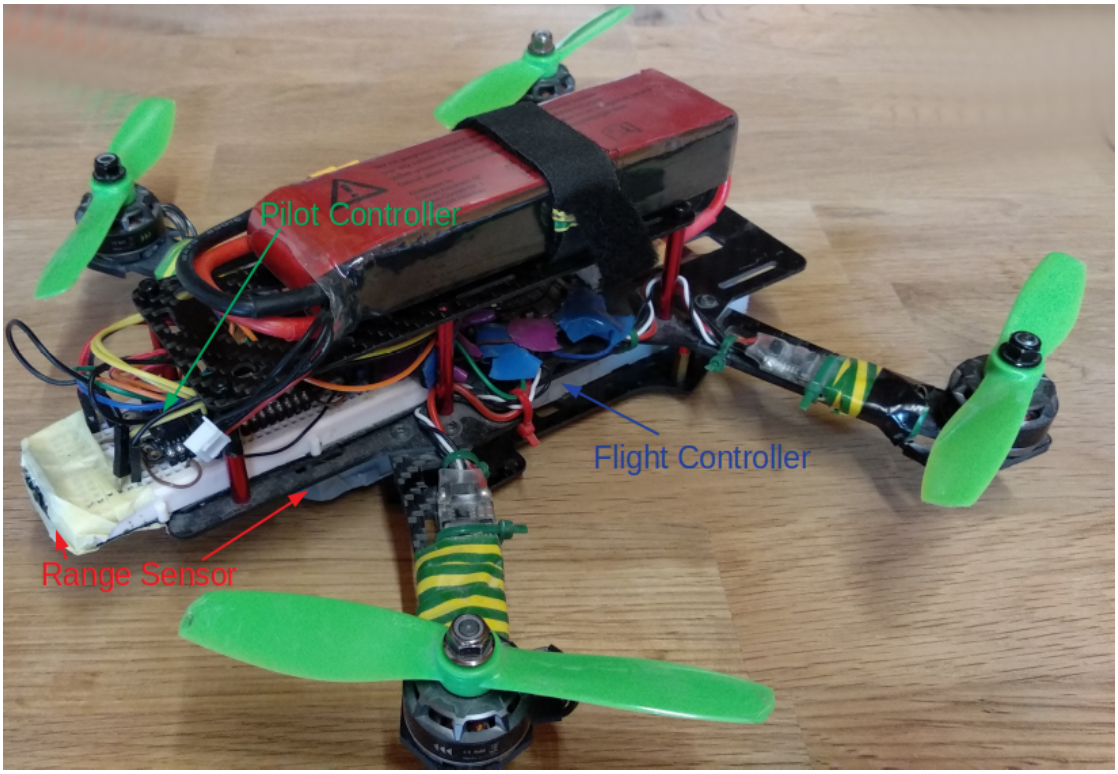


Figure 4.10: Final Build

CHAPTER 5

Software of the Quadcopter

In this chapter we will discuss the developed software as well as the configuration of existing software.

5.1 Configuration of Initial Build

For the initial build we used already existing parts (Section 4.2) therefore we only need to configure the parts. We used as flight controller the *Matek F405-STD* with *Betaflight 3.2.2* installed. To configure *Betaflight* the *Matek F405-STD* can be connected to a PC over Universal Serial Bus (USB). Figure 5.1 and Figure 5.2 show the configuration done. The Firmware used on the ESC's is *BLHeli* 14.9. We can use the *BLHeli-Configurator* to configure each ESC. The tool uses the *Betaflight* pass through feature where data is passed through the flight controller to each ESC. This is handy because we do not need to connect each ESCs by its own to the PC. The *BLHeli-Configurator* can configure all ESC at once. The used configuration is shown in Figure 5.3. One important feature is the DampedLight. With this feature the ESC breaks the motor active on throttle decrease.

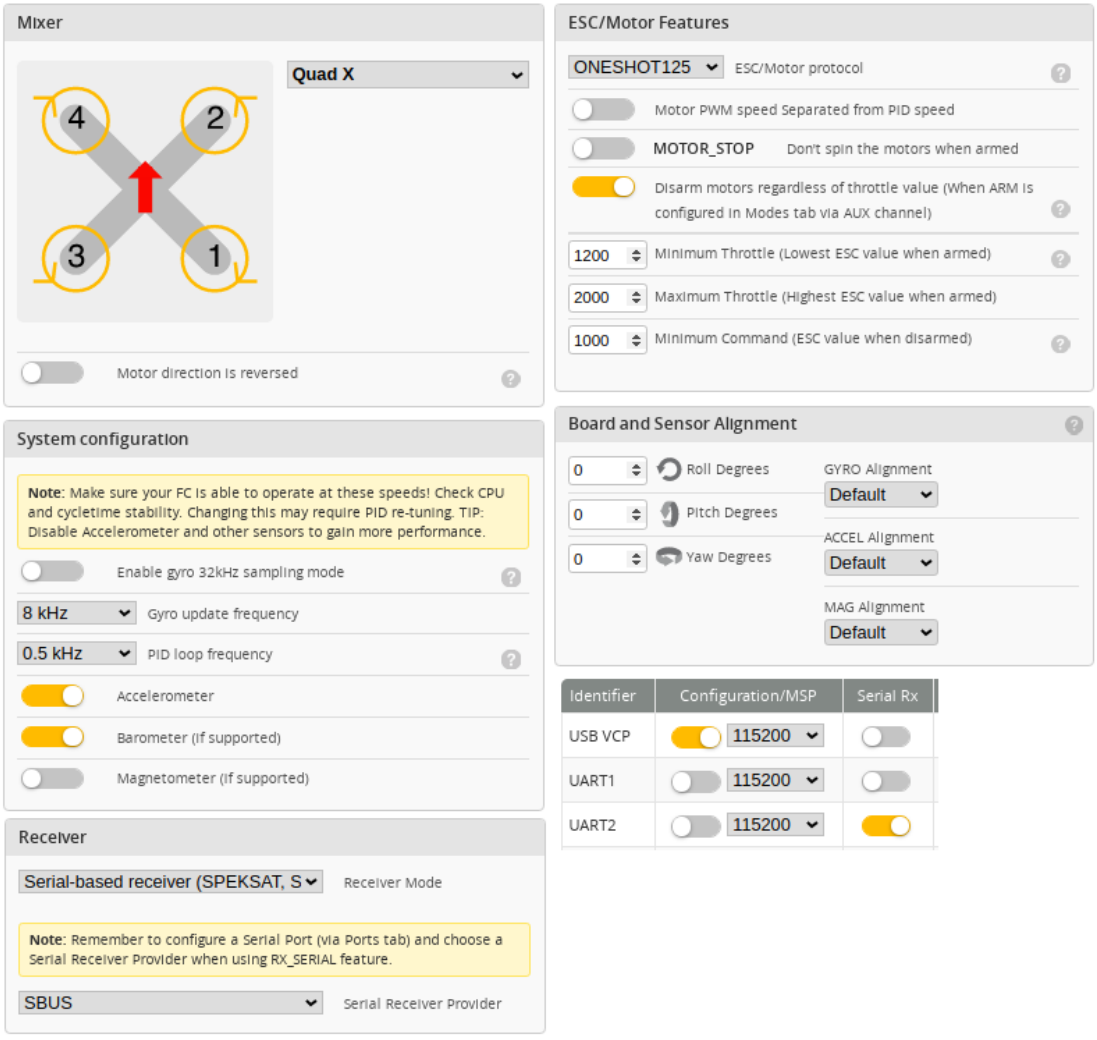


Figure 5.1: *Betaflight-Configurator* settings for *Matek F405-STD*.

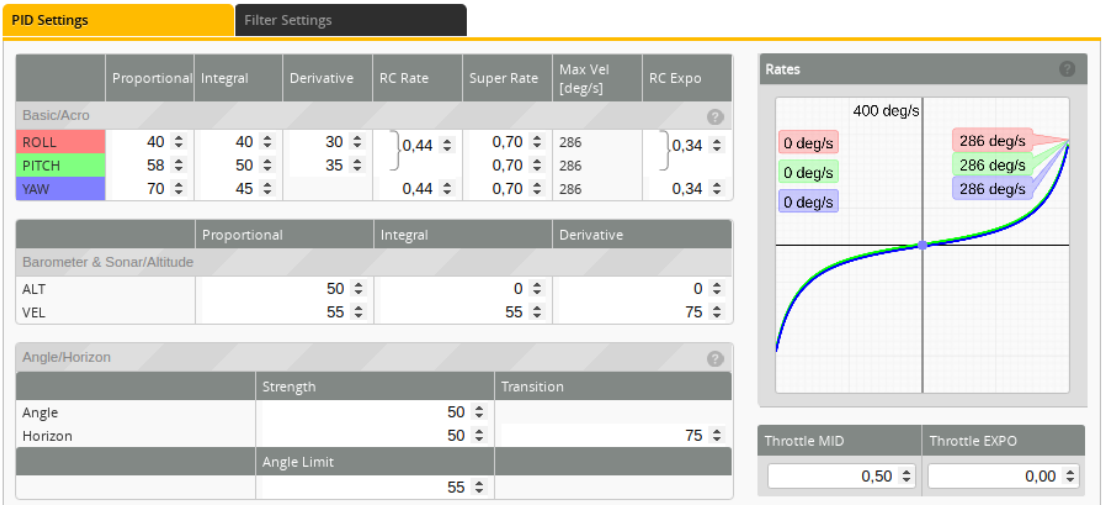


Figure 5.2: *Betaflight-Configurator* PID settings for *Matek F405-STD*.

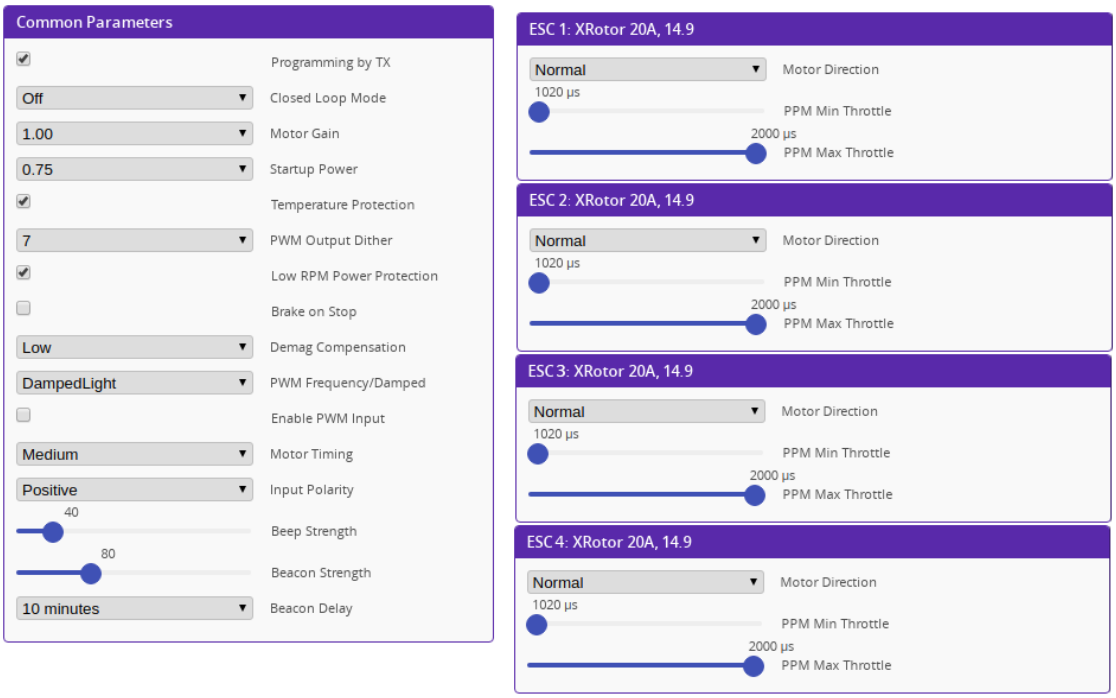


Figure 5.3: ESC settings configured via *BLHeli-Configurator*.

5.2 Software for Altitude Hold Build

For the Altitude Hold Build (see Figure 4.5 for the schematic of the hardware) we developed a gateway based on the *Simblee RFD77101*. The main task of the gateway is to get data from PC and translate it to data the flight controller understands as well as sending back telemetry data from the flight controller to the PC.

5.2.1 PC

On the PC (Ubuntu 18.04) ROS is used to manage the data exchange between a joystick node and a gateway node.

- **Joystick node**

  This node gets data from an joystick connected to the PC and publishes it for other nodes. We use the Play Station (PS)3 controller because this is already supported by ROS and Ubuntu.

- **Gateway node**

  To send/receive data to/from the gateway BLE is used. This is done by the python tool *pygatt*. This tool uses the *BlueZ* driver from Linux to interact with the bluetooth hardware on the PC. The gateway node supports a *level mode* and *hold mode*. In *level mode* the thrust is controlled by the joystick, in *hold mode* the thrust gets calculated based on the range sensor value.

5.2.2 Simblee Gateway

The *Simblee RFD77101* is a SiP with *Bluetooth Smart* radio transceiver with a built-in ARM Cortex M0 MCU. This MCU can be programmed using the Arduino IDE and the Simblee extension. This makes it easy to program but makes it less flexible if some extra features are necessary. The gateway has two important hardware interfaces. These interfaces are using different protocols as shown in Figure 5.4.

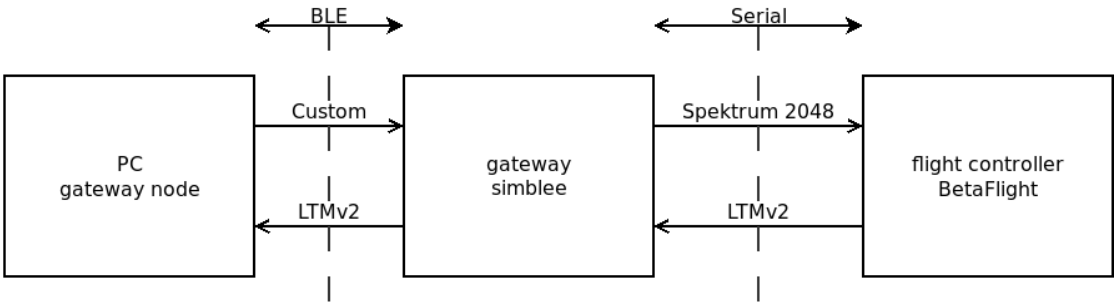


Figure 5.4: Interfaces between PC, *Simblee* and flight controller.

From the PC to the gateway BLE is used as interface. With this interface we use no particular protocol to send data to the gateway. The data sent to the gateway is shown in Figure 5.5a. The connection between the gateway and the flight controller is done by UART. The gateway sends control data to the flight controller using the *Spektrum 2048* protocol [Hor]. The received data from the flight controller uses the LightTelemetry v2 (LTMv2) protocol. This protocol is also used to send data back to the PC. All LTMv2 frames from the flight controller are buffered and then sent unchanged to the PC. The gateway also adds two custom frames. The L frame includes the range value from the TOF sensor and is sent every time the sensor gets a new value. The Z frame includes system statistic about the gateway node. *Time used* and *time total* can be used to estimate the CPU usage of the gateway. The Spektrum 2048 protocol expects every 11ms a new frame but the BLE can be slower so the gateway tracks with *missed* and *success* the number of Spektrum 2048 frames which can not be sent to the flight controller. If too many frames are missed the flight controller gets in a fail save state (can be configured in *Betaflight*) we use 400ms which are about 37 frames. All information in the Z Frame is based on events between two Z Frames. Z Frames are sent every second from the gateway to the PC. One important interface is the connection to the range sensor which uses the I2C protocol for the configuration and ranging. Fortunately there is an Arduino library provided from Pololu [Pol] which does all this for us. We configure the sensor to a range up to 2 meters and a sensing interval of 33ms (long range profile). The sensor also supports an interrupt driven sensing. Which means it indicates on the GPIO pin that new data is available. So we only need polling the pin and not polling over the interface.

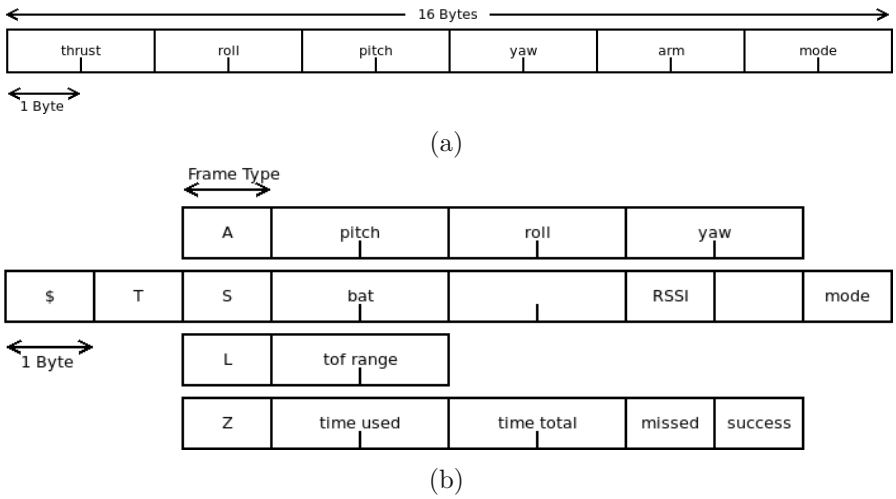


Figure 5.5: In (a) the frame sent from PC to the Simblee gateway is shown, in (b) the frames sent from the gateway to the PC are shown.

5.3 Flight Controller Firmware: Betaflight Port Attempt

This firmware is an attempt to port *Betaflight* to the *BMF055*. We choose version 4.1 of *Betaflight* because *Betaflight* introduces in this version a Hardware Abstraction Layer (HAL) to the firmware. The port of *Betaflight* is then done by a feature/interface approach where we port step by step more features/interfaces from *Betaflight* to the *BMF055*. For building the firmware we use the same build system as *Betaflight* by introducing the *samd20j18* as new MCU to the system and only adding the ported features/interfaces.

The first feature ported was the serial interface so we can communicate with the flight controller over MSP. Because Betaflight supports dynamic binding of serial ports over the *Betaflight Configurator* this was more complicated then expected. We then ported the IMU feature and the sensors/receiver interface. For the receiver interface we implemented the Spektrum2048 protocol.

Because the code quality from the ported features/interfaces is not great and the time it takes to port such a feature/interface we decided to discard the approach to port *Betaflight* and take the missing features for this firmware (pid controller and scheduler) from Blocher's work [Blo]. The source code for the firmware can be found here [Temd].

5.4 Flight Controller Firmware

During the development of the collision avoidance we encounter the problem that the *Betaflight Port Attempt* does not generate enough debug data to localize problems. Therefore, we developed a custom firmware where we aimed for better testing and debugging. The source code for the firmware can be found here [Tema].

We adapt and reuse some modules from the *Betaflight Port Attempt* so ideas and naming conventions are based on *Betaflight* [Beta] and Blocher's work [Blo]. The hardware abstraction is done by defining interfaces for each module which needs access to hardware components. These interfaces define callbacks which have to be implemented by platforms the firmware is running. We define the **BMF055** and **SITL** platform as targets for this firmware. For detailed information about the implementation of the modules see Appendix A.3. Here we will discuss the hardware independent modules (see modules above the HAL in Figure 5.6).

- **Time**
  This module provides the time base to the flight controller.
- **Configuration**
  The module provides the configuration like **MINTHROTTLE** (minimal motor command for the ESC) parameter to the flight controller.

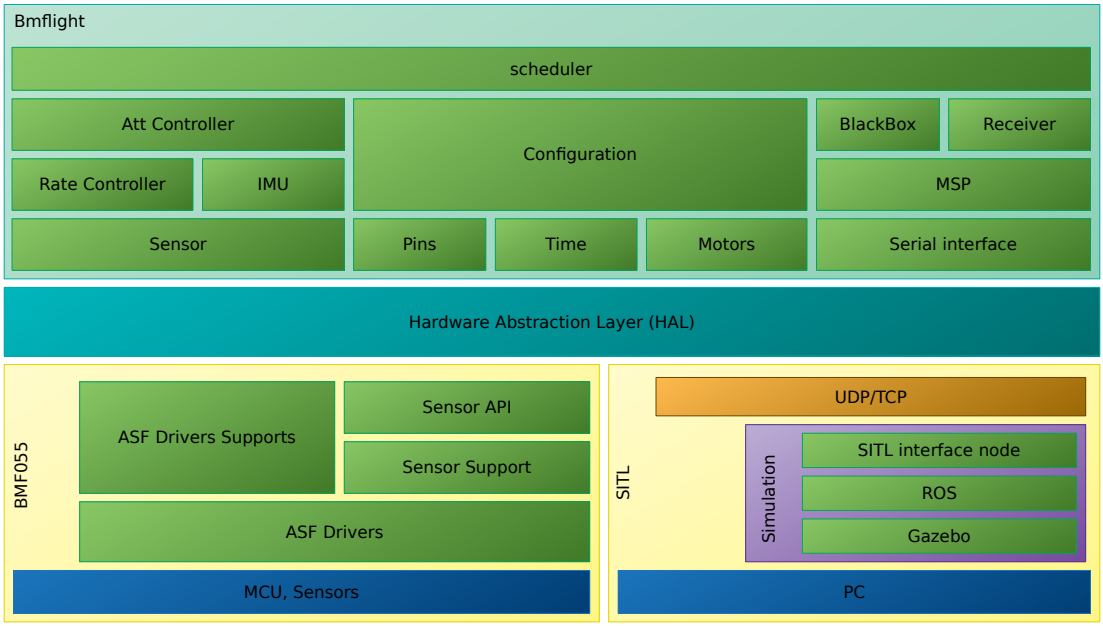


Figure 5.6: Flight controller firmware layers.

- **Sensor**
  This module provides methods to request new data from the sensor hardware. The firmware supports an accelerometer and a gyroscope.

  – **Accelerometer**
  To get a new value the *updateACC* method can be called. This function first calls the sensor driver to get a new value from the sensor. Then, the offset is compensated and the scale is applied. The value of the offset compensation of the accelerometer is done on raw values of the sensor and stored in the configuration parameter ACC_TRIM. This parameter can be generated automatically via the *Betaflight Configurator* tool. If the calibration is triggered the flight controller calculates the average of 500 successive readings (make sure the quadcopter stands still on a horizontal surface during the calibration). The offset of the z-axis will be corrected by the earth gravitation.

  – **Gyroscope**
  To get a new value the *updateGyro()* can be called. This method also performs the calibration of the sensor during first one thousand readings. During this time (2 seconds on 500Hz looptime) the quadcopter must not be moved. The calibration restarts if the quadcopter was moved in this time.

- **Inertial Measurement Unit**
  This module abstracts the IMU algorithm with the *imu* interface. The module will call the *update* method of the chosen estimator each time the *updateEstimatedAttitude* method gets called. We used the *Betaflight* IMU algorithm which is based on the MahonyAHRS algorithm. For more information see [Mad] and [Beta].

5.4.1 Rate-Controller

In Section 2.3 we discussed that controlling a quadcopter is done by a cascading of controllers. We also use this approach for the developed firmware. Figure 5.7 shows the cascading controllers used by the flight controller and the *Pilot Controller*.

The *Rate-Controller* is implemented as PID-Controller. In Equation 2.20 and 2.19 we described the PID-Controller time continuous. To implement the controller we use the time discrete form (5.1 and 5.2) of the equations.

$$e[n] = x[n] - y[n] \tag{5.1}$$

$$u[n] = K_P\, e[n] + K_I \sum_{k=1}^{n} e[k]T + K_D \frac{e[n] - e[n-1]}{T} \tag{5.2}$$

Because the *samd20j18a* does not support float operations in hardware and we run the *Rate-Controller* at 500Hz we use integer arithmetic for the calculations to meet these time constrains. As in Figure 5.7 shown the *Rate-Controller* outputs a vector (limited to a range [−500,500])

$$\boldsymbol{\rho} = \begin{bmatrix} \rho_\phi \\ \rho_\theta \\ \rho_\psi \end{bmatrix} = \begin{bmatrix} \rho_{p\,\phi} + \rho_{i\,\phi} + \rho_{d\,\phi} \\ \rho_{p\,\theta} + \rho_{i\,\theta} + \rho_{d\,\theta} \\ \rho_{p\,\psi} + \rho_{i\,\psi} + \rho_{d\,\psi} \end{bmatrix} \tag{5.3}$$

which will be picked up later by the motor mixer to create the signals for the ESC's. The calculation at time n is done independently for each axis. First the error $e[n]$ between desired angular velocity $\omega_{Bs}[n]$ and measured velocity $\omega_B[n]$ is calculated.

$$e[n] = \omega_{Bs}[n] - \omega_B[n] \tag{5.4}$$

$\omega_{Bs}[n]$ and $\omega_B[n]$ are defined in deci degrees per second. The proportional term of the output $\rho_p[n]$ is calculated for each axis with

$$\rho_p[n] = \frac{kp\; e[n]}{256}\,. \tag{5.5}$$

The integrating term $\rho_i[n]$ is calculated with

$$\rho_i[n] = \frac{e_i[n]}{8192}\,. \tag{5.6}$$

The integration error $e_i[n]$ is protected by a saturation to counteract a wind up effect. (Δt is the time between two samples)

$$e_i[n] = e_i[n-1] + e[n] \left(\frac{\Delta t}{2048}\right) ki \tag{5.7}$$

the factor $\left(\frac{\Delta t}{2048}\right)$ is for normalization.

$\rho_d[n]$ is calculated with a moving average to smooth the output:

$$\Delta_e[n] = e[n] - e[n-1] \tag{5.8}$$

$$\Delta_s[n] = \Delta_e[n] + \Delta_e[n-1] + \Delta_e[n-2] \tag{5.9}$$

$$\rho_d[n] = \frac{\Delta_s[n]\ kd}{256} \tag{5.10}$$

Note that we don't need T (defined in Equation 5.2) because we assume it to be always the same, therefore we can move it to scaling and kd. The 1/3 of the average in Equation 5.9 is also moved in the scaling factor in Equation 5.10.

5.4.2 Att-Controller

The attitude controller was implemented as a P-Controller where the controls $\boldsymbol{\eta}_{c\,a}$ reflects the set point for the roll ϕ_s and pitch θ_s angle of the quadcopter. The yaw is already the angular velocity in r_s and gets passed by to the *Rate-Controller*.

The $\boldsymbol{\omega}_{Bs}$ (in deci degrees per second) is now calculated by:

$$\boldsymbol{\omega}_{Bs} = \begin{bmatrix} p_s \\ q_s \\ r_s \end{bmatrix} = \begin{bmatrix} l\ \phi_s \\ l\ \theta_s \\ r_s \end{bmatrix} \tag{5.11}$$

where

$$l = \frac{\omega_{limit}}{500}\ . \tag{5.12}$$

ω_{limit} is the maximum angle allowed for ϕ, θ which is defined with **deciLevelAngleLimit** in the configuration. l includes the normalization back to range $[-500, 500]$.

Figure 5.7: Controller Structure

5.4.3 Serial Interface

We define the *serialPort* interface for abstracting the hardware. It is possible to define more than one *serialPort* by multiple instances of this interface. Each instance need the following callback/data for initialization:

- ***txBuffer*** pointer to the transmit buffer
- ***txBufferSize*** size of the transmit buffer
- ***rxBuffer*** pointer to the receive buffer
- ***rxBufferSize*** size of the receive buffer
- ***serialWrite*** write one byte to the *serialPort*
- ***serialRead*** read one byte from the *serialPort*
- ***serialTotalRxWaiting*** number of bytes ready to be read from receive buffer
- ***serialTotalTxFree*** number of bytes free in transmit buffer
- ***isSerialTransmitBufferEmpty*** true if no bytes need to be transmitted
- ***beginWrite*** lock the *serialPort* to buffer some data before submission starts
- ***endWrite*** unlock the *serialPort* and start the transmission of the buffered data

The *serialPort* handles sending and receiving data with two ring buffers. The **serialWrite** method writes the data to the txBuffer and by default starts transmission. With **beginWrite** it is now possible to lock this behavior to buffer some data before transmission. By calling **endWrite** the transmission can then be restarted.

On the **BMF055** we used one Universal Synchronous and Asynchronous Serial Receiver and Transmitter (USART). On **SITL** we used TCP without the lock mechanism (beginWrite, endWrite functions).

5.4.4 MSP Port

The **mspPort** is based on the serial interface. All reading and writing of data is handled by the serial interface. The encoding and interpreting of the data from the serial interface is handled by **mspPort** by calling the *mspProcess* function. The **mspPort** is reading data from the serial interface until no data is available or a MSP message is decoded. For decoding incoming frames the state machine in Figure 5.8 is used (inspired by the implementation done by *Betaflight* [Betd]).

The states *MSP_HEADER_START*, *MSP_HEADER_M* and *MSP_HEADER_V1* are for decoding the header of the MSP frame. On idle the *mspPort* is in the state *MSP_IDLE* if a '**$**' is read from the *serialPort* the state is changed to *MSP_HEADER_START*. In *MSP_HEADER_START* the offset and checksum of the *mspPort* are set to zero. If the next char received is '**M**' the state transition to *MSP_HEADER_M*. On each other char we transition back to *MSP_IDLE*. If the *mspPort* is in state *MSP_HEADER_M* and '>' is received the *mspPort* transition to *MSP_HEADER_V1* and set the packetType of the *mspPort* to *MSP_PACKET_REPLY*. If the next char received is '<' we also transition to *MSP_HEADER_V1* but set the packetType of the *mspPort* to *MSP_PACKET_COMMAND*.

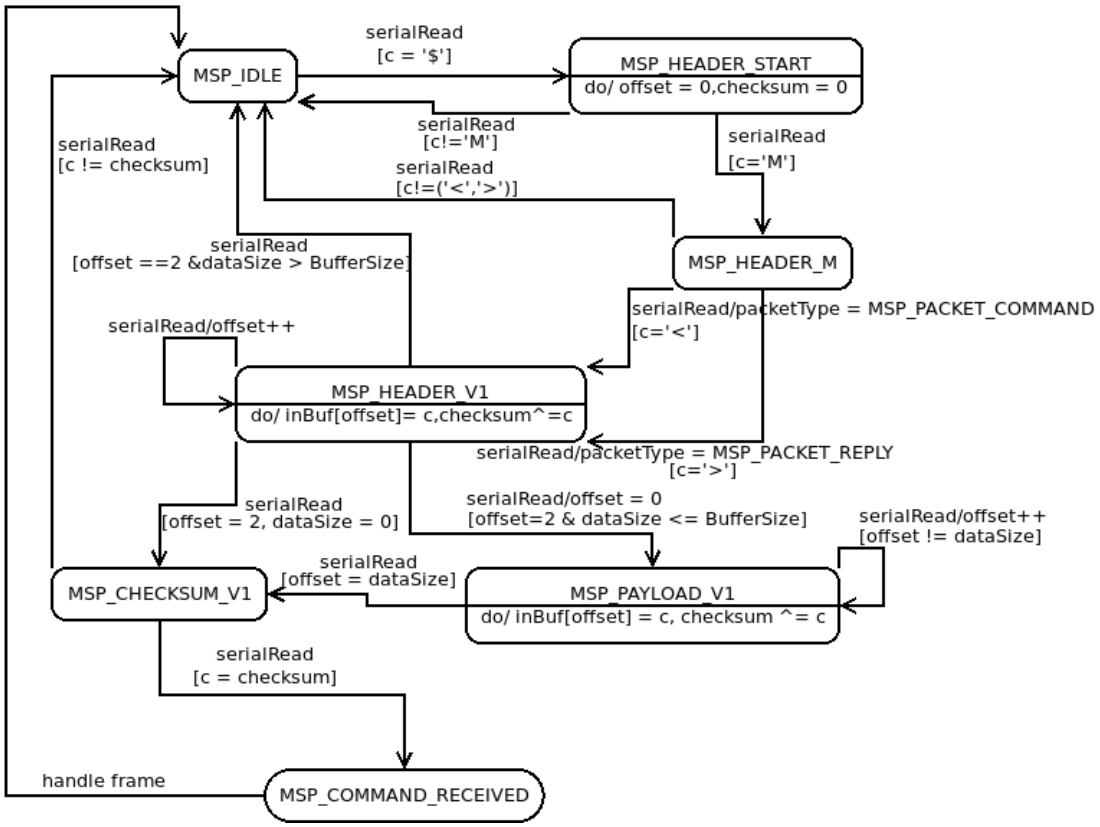


Figure 5.8: State machine for decoding MSP frames.

In *MSP_HEADER_V1* the dataSize and MSP command type is parsed. If the parsed dataSize exceeds the input buffer of the *serialPort* we transition back to *MSP_IDLE*, otherwise we transition to *MSP_CHECKSUM_V1* (if no data needs to be parsed) and to *MSP_PAYLOAD_V1* if data needs to be parsed. We stay in the state *MSP_PAYLOAD_V1* until dataSize chars are read and then transition to *MSP_CHECKSUM_V1*. The next char received on the *serialPort* in state *MSP_CHECKSUM_V1* is the checksum of the frame. If the read data is equal to the calculated checksum we transition to *MSP_COMMAND_RECEIVED* otherwise back to *MSP_IDLE*. The *mspPort* stays in state *MSP_COMMAND_RECEIVED* until the decoded frame was processed.

The data parsed by the *mspPort* is now wrapped in to a mspPacket holding all information for further processing the data. If the decoded frame is a MSP response frame (packetType=*MSP_PACKET_REPLY*) the *mspProcessReplyFnPtr* function (defined in the *mspPort* interface) is called. The Firmware does nothing in this function because the quadcopter acts as an MSP slave. This means it will not send request or command frames and therefore does not expect responses (nevertheless we used the interface also for the *PC Gateway Tool* where the function was indeed defined).

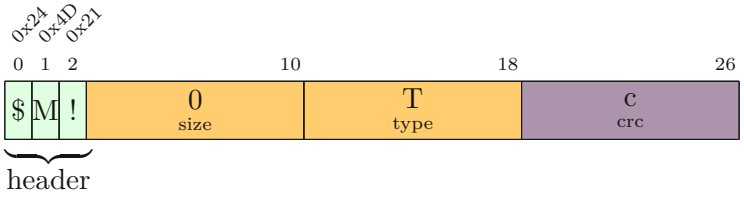


Figure 5.9: MSP error extension frame.

If the decoded frame is a MSP request frame (packetType=*MSP_PACKET_COMMAND*) the *mspProcessCommandFnPtr* function (defined in the *mspPort* interface) is called with the decoded data as command input and a mspPaket where the function can put the reply to the command. After *mspProcessCommandFnPtr* the reply mspPaket will be sent over the serial interface. It is possible to suppress an reply by setting the result status in the reply mspPaket to *MSP_RESULT_NO_REPLY*.

The firmware supports different command types. If a command is not supported the firmware returns an error frame. This error frame is an extension of the MSP. In Figure 5.9 the error frame is shown, the type is always the requested (not implemented) command type, also the extension is backwards compatible because "!" is invalid.

5.4.5 Task and Scheduler

A task is defined by:

- ***taskName*** name of the task
- ***taskFunc*** function pointer to the function the task executes
- ***priority*** priority of the task (higher value == higher priority)
- ***desiredPeriodUs*** the desired period time the task should be executed in microseconds

On startup a task gets enqueued in a list of tasks. This list is sorted by the ***priority*** of the tasks. The tasks defined by the firmware are listed in Table 5.1. The scheduler used is a non preemptive scheduler, this means a task cannot interrupt another task.

Table 5.1: List of Tasks defined by the flight controller firmware.

taskName	**taskFunc**	**priority**	**desiredPeriodUs** (μs)
TASK_LOOP	taskLoop	200	2000
TASK_ATTITUDE	taskAttitude	100	4000
TASK_SERIAL	taskHandleSerial	4	4000
TASK_RX	taskRx	3	10000
TASK_SYSTEM	taskSystemLoad	1	100000
TASK_LED	taskLed	0	200000
TASK_DEBUG	taskDebugSerial	0	250000

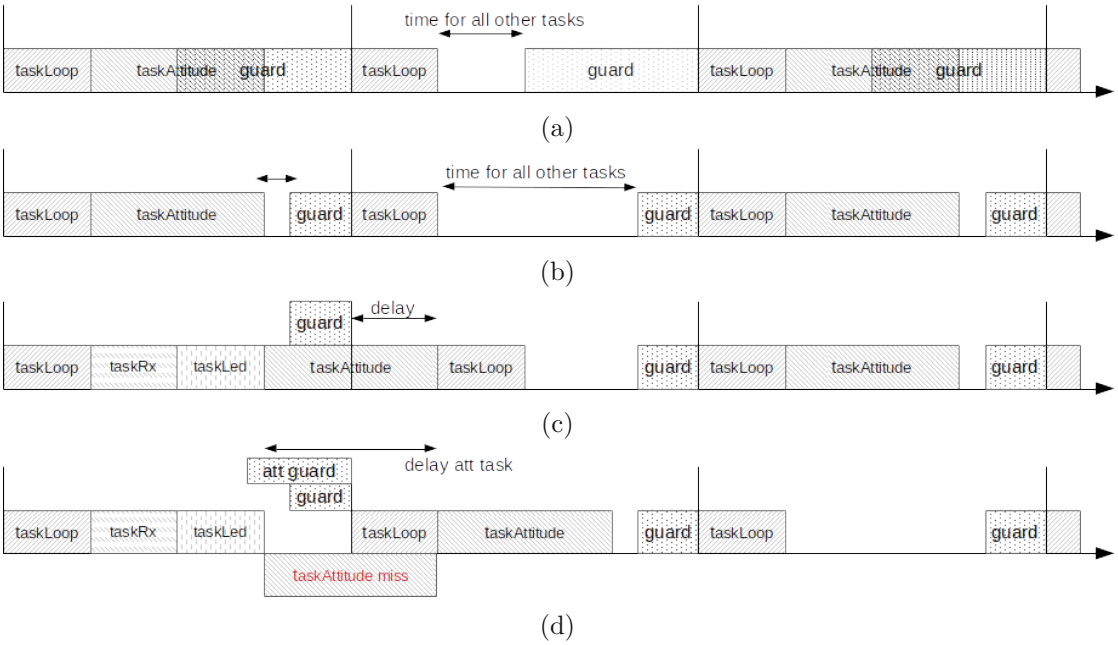


Figure 5.10: In (a) the timing with an optimal guard is shown. In (b) the impact of a reduced guard window is shown. (c) shows the delay of the taskLoop if another long task gets scheduled. In (d) the attitude guard window is shown.

We implement two different scheduler strategies and compare them on how well they are able to meet the desired period of the *TASK_LOOP* and *TASK_ATTITUDE* (these two tasks are mainly responsible for the stability of the quadcopter)

Dynamic Scheduler

This strategy is a simplified version of the scheduler used by *Betaflight* [Beta]. The scheduler dispatches tasks based on their priority. Tasks with higher priority get dispatched first if two or more tasks are ready for next execution. This strategy has the disadvantage that tasks with lower priority never get dispatched if higher priority tasks need too much CPU time. It can also happen that the scheduler dispatches a low priority task just before a high priority task runs and therefore the high priority task misses its deadline. Because controlling a quadcopter is a real time application we need adaption to this dynamic strategy to ensure the *TASK_LOOP* is not delayed. First we define the *TASK_LOOP* as *realtime* task. This means the scheduler will always try to dispatch this task first (this is the same as giving this task the highest priority). Second we guard the *TASK_LOOP* by a time window t_G. This means if the *TASK_LOOP* should be dispatched at time t_d we will no longer dispatch any task if

$$t_{current} >= (t_d - t_G) \; . \tag{5.13}$$

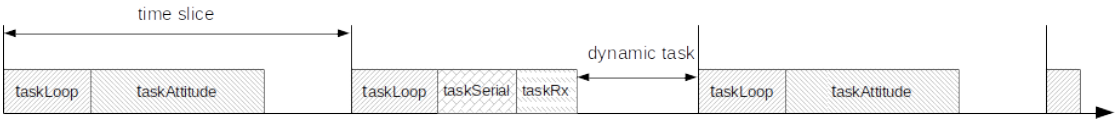


Figure 5.11: Static scheduler timing.

If we want that the *TASK_LOOP* is never delayed the window t_G has to be the time the longest task needs to be executed. In our application this is the task *TASK_ATTITUDE* with about 1200μs. But this leads to the problem that we only have a small window where all other tasks can be executed, see Figure 5.10a. To prevent this we decrease the guard window, see Figure 5.10b. This can lead to problems if other tasks are executed before *TASK_ATTITUDE* gets ready, see Figure 5.10c which results in a delay of *TASK_LOOP*. To counteract this problem we specified for *TASK_ATTITUDE* an extra guard for *TASK_LOOP*. This only means *TASK_ATTITUDE* will not be dispatched if not enough time is left to complete, see Figure 5.10d. This can result in a delay for *TASK_ATTITUDE*.

Static Scheduler

Because the time constraints are so tight we decided to use a static scheduler. This scheduler is based on time slices as shown Figure 5.11. Each time slice starts with *TASK_LOOP* after that *TASK_ATTITUDE* or *TASK_SERIAL* and *TASK_RX* are executed and if time remaining we dispatch all task dynamically based on the priority if they are ready.

5.4.6 Motor Control

The *Motor Control* is responsible for mixing the *Rate-Controller* output and outputs the corresponding signals to the *Motor Driver*. The main idea is ported from [Bas].

Mixing

The *Rate-Controller* output is defined on roll, pitch and yaw axis of the quadcopter. We now have to translate these commands to individual commands for each motor.

The mixer takes as input a vector

$$\vec{m_{input}} = \begin{bmatrix} i_t \\ \rho_\phi \\ \rho_\theta \\ \rho_\psi \end{bmatrix} \tag{5.14}$$

where i_t is the current throttle value in the range of $[1000, 2000]$ and ρ_ϕ, ρ_θ, ρ_ψ are the outputs of the *Rate-Controller* in the range of $[-500, +500]$.

Using the quadcopter in symmetric X-Quad configuration (see Figure 5.12b) results in the transformation matrix:

$$M_{sQuad} = \begin{bmatrix} \vec{m}_t & \vec{m}_r & \vec{m}_p & \vec{m}_y \end{bmatrix} = \begin{bmatrix} 1 & -1 & 1 & 1 \\ 1 & -1 & -1 & -1 \\ 1 & 1 & 1 & -1 \\ 1 & 1 & -1 & 1 \end{bmatrix} . \tag{5.15}$$

The vector $\vec{m}_t$ describes how much from the throttle is mixed to the motor. This is useful if a build uses different motors (small differences regulated out by the *Rate-Controller*).

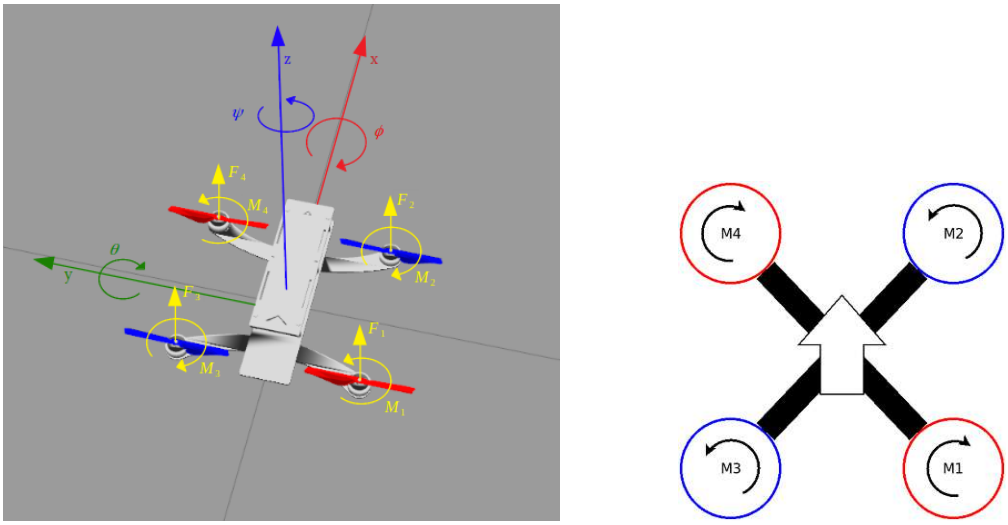
Figure 5.12a shows that if the pilot wants to roll right we need the motors 1 and 2 decrease and motors 3 and 4 increases the rotational speed. Therefore, the vector $\vec{m}_r$ in the transformation matrix M_{sQuad} is defined as

$$\vec{m}_r = \begin{bmatrix} -1 \\ -1 \\ 1 \\ 1 \end{bmatrix} . \tag{5.16}$$

Analogous the pitch of the quadcopter is defined by $\vec{m}_p$ here the motors 1 and 3 increase and the motors 2 and 4 decrease the rotational speed.

The physical background of yaw is defined with Equation 2.12. Each propeller generates an angular momentum for the quadcopter which is in opposite direction of the rotation direction (Newtons 3rd law). If the two diagonal propellers spin in the same direction and the other two diagonal in the other direction the sum of the generated momentum is then equal to zero. The vector $\vec{m}_y$ therefore sums up to zero. The direction of the values in the vector is now depending on the spin direction of the motors. In our configuration motor 1 and motor 4 are spinning clockwise (generating a positive momentum around ψ) and motor 2 and motor 3 are spinning counter clockwise (generating a negative momentum around ψ). So if we want to rotate to the left (positive around ψ) motor 1 and 4 need to increase and motor 2 and motor 3 need to decrease therefore

$$\vec{m}_y = \begin{bmatrix} 1 \\ -1 \\ -1 \\ 1 \end{bmatrix} . \tag{5.17}$$



(a) (b)

Figure 5.12: In (a) the rotation axes of the quadcopter and generated momentum of each motor are shown. (b) shows the X-Quad configuration of the quadcopter.

Limits and Configurations

After mixing the *Rate-Controller* output to the motor values M_{out} we have to make sure that the produced values do not exceed the ESC values. The value range for the ESC signals is $[MINTHROTTLE, MAXTHROTTLE]$. It can happen that a motor exceeds its maximum. For example if we use a high throttle ($i_t = 1900$) and want the quadcopter also to roll left ($\rho_\phi = -500$) we need to spin motor 1 and 2 faster. For Motor 1 the equation is (if we only want to roll):

$$\begin{aligned} m_1 &= m_{t1} * i_t + m_{r1} * \rho_\phi + m_{p1} * \rho_\theta + m_{ty} * \rho_\psi \\ &= 1 * 1900 + (-1) * (-500) + 1 * 0 + (-1) * 0 \\ &= 2400 \end{aligned} \tag{5.18}$$

which exceeds $MAXTHROTTLE$ of 1980. If we set all values that exceeds the maximum to $MAXTHROTTLE$ we will get that the quadcopter in our example will roll very slow or nothing at all. This is called a *bad gyro correction* of the quadcopter. This means the quadcopter can not achieve a desired angular velocity. To get a better gyro correction we can dump all motors, not only the one which was exceeding the maximum, about the amount we exceeded the maximum:

$$m_{max} = MAX(M_{out}) \quad , \quad m_{correct} = m_{max} - MAXTHROTTLE \tag{5.19}$$

$$M_{out} = \begin{bmatrix} m_1 - m_{correct} \\ m_2 - m_{correct} \\ m_3 - m_{correct} \\ m_4 - m_{correct} \end{bmatrix} = \begin{bmatrix} m_1 - (m_{max} - MAXTHROTTLE) \\ m_2 - (m_{max} - MAXTHROTTLE) \\ m_3 - (m_{max} - MAXTHROTTLE) \\ m_4 - (m_{max} - MAXTHROTTLE) \end{bmatrix} \tag{5.20}$$

The same is also true for $MINTHROTTLE$. It can happen that a motor falls below $MINTHROTTLE$. In this case we can also do the same:

$$m_{min} = MIN(M_{out})\ quad, \quad m_{correct} = MINTHROTTLE - m_{min} \tag{5.21}$$

$$M_{out} = \begin{bmatrix} m_1 + m_{correct} \\ m_2 + m_{correct} \\ m_3 + m_{correct} \\ m_4 + m_{correct} \end{bmatrix} = \begin{bmatrix} m_1 + (MINTHROTTLE - m_{min}) \\ m_2 + (MINTHROTTLE - m_{min}) \\ m_3 + (MINTHROTTLE - m_{min}) \\ m_4 + (MINTHROTTLE - m_{min}) \end{bmatrix} \tag{5.22}$$

We also defined a $MINCHECK$ for the throttle. If the throttle is lower than this value we output $MINCOMMAND$. If the quadcopter is not armed (ready to fly) we also output $MINCOMMAND$.

5.4.7 Blackbox

This module is optional and can be enabled/disabled on pre-processor level by the define *USE_BLACKBOX*. It is responsible for logging the raw internal states of the quadcopter during flight to enable analysis, debugging and tuning of the quadcopter. To be compatible with *Betaflight Blackbox Explorer* we ported the code from *Betaflight* [Beta].

Often the logs are written to a SD-Card by the flight controller. Because we are using the *Decentralized Approach* we can hand over the storage of the data to the *Pilot Controller*. Therefore, we introduce additional MSP commands for transmitting the data from the flight controller to the *Pilot Controller*. It is possible to modify the data which will be logged on pre-processor level by the defines in *blackbox.c*. The general structure of a log file is shown in Figure 5.13a. Within the log header a few fields are required so that the *Betaflight Blackbox Explorer* can decode the stored data (see Figure 5.13b).

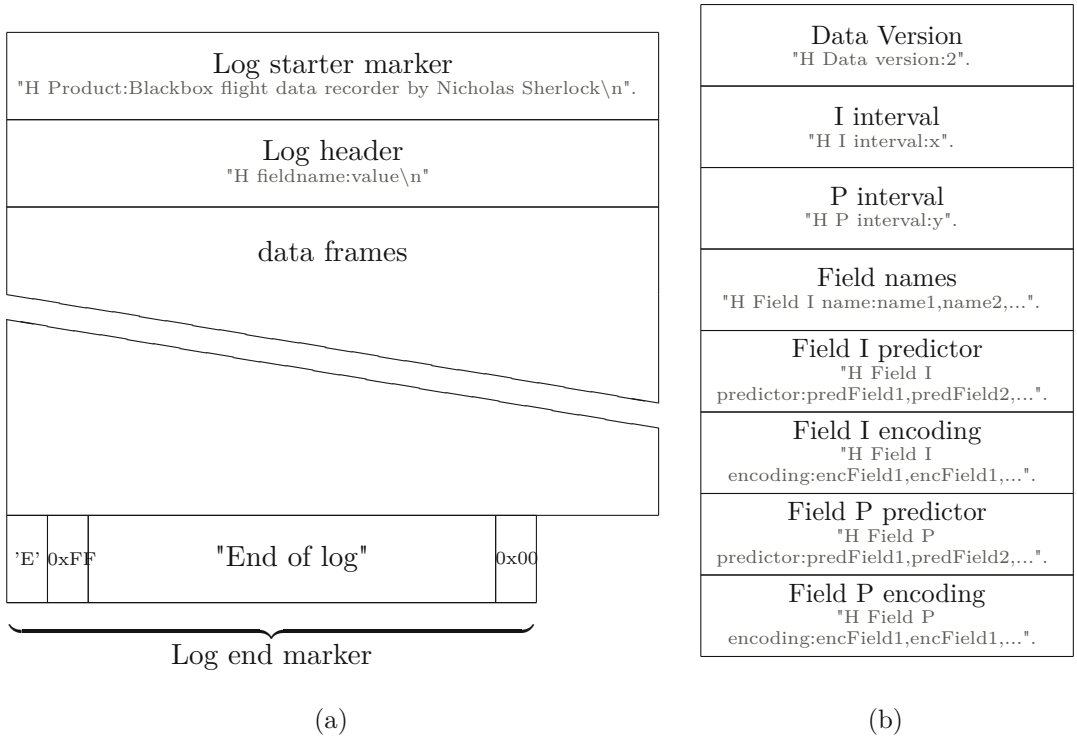


Figure 5.13: In (a) the general log file structure is shown, (b) shows details of the log file header structure.

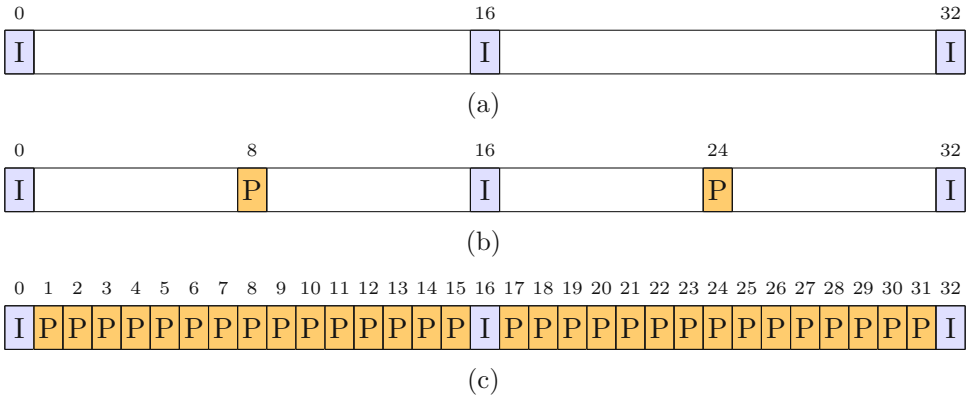


Figure 5.14: Blackbox *P-interval* examples. In (a) the *P-interval* is set to = 1/16, in (b) to = 1/8 an in (c) to = 1/1.

For features like the Fast Fourier transform (FFT) the extra headers *minthrottle*, *maxthrottle*, *gyro_scale*, *acc_1G* and *looptime* need to be defined. The data is encoded and compressed in *data frames*. We use the main frames *I-* and *P-frames* for logging the main states of the quadcopter (PID internals, sensor, motor , rcCommand and debug data). Like in video compression the *I-frames* ("intra" frames) can be decoded without any other data frame. For decoding a P-frame the predecessor frame of the P-frame is needed because only the differences to the predecessor frame are encoded in a P-frame. The ratio of these two frames is defined in the header file (see Figure 5.13b). The *I interval* defines how often the main loop (in our case *TASK_LOOP*) will write an *I-frame*. If the main loop is running with $500Hz$ and *I-interval* equals to 16 we get every $32ms$ a new *I-frame*. The *P-interval* defines how often a *P-frame* is logged between two *I-frames* (see Figure 5.14). More information on encoding and predictor is provided by [Bete]. In Table 5.2 the used encoding and predictor are shown. If the blackbox is enabled, logging is started when the quadcopter gets armed and ends on disarming the quadcopter. For initializing a log with the proper header we use a state machine (see Figure 5.15) because writing all header data at once would exceed the serial buffer.

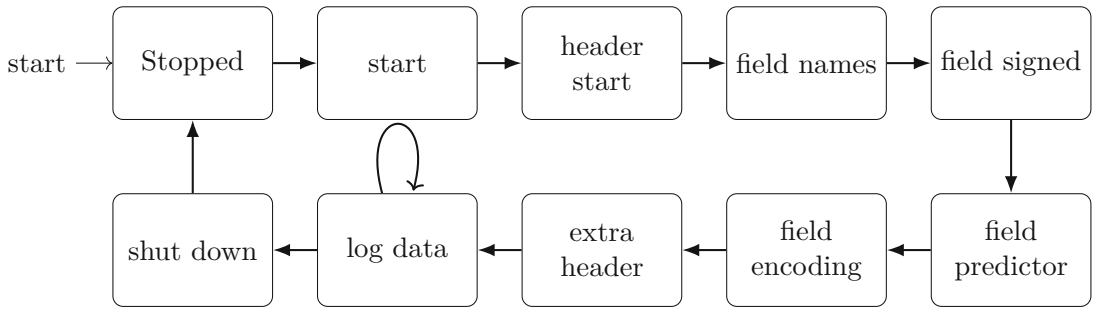


Figure 5.15: Blackbox state machine for writing log files.

Table 5.2: Used Blackbox predictor and field encoding.

encoding:

- **0** Signed variable byte
- **1** Unsigned variable byte
- **7** TAG2_3S32
- **8** TAG8_4S16
- **9** NULL

predictor:

- **0** Predict zero
- **1** Predict last value
- **2** Predict straight line
- **3** Predict average 2
- **5** Predict motor[0]
- **6** Predict increment
- **4** Predict *minthrottle*

Field	encoding I	predictor I	encoding P	predictor P	signed
loopIteration	1	0	9	6	0
time	1	0	0	2	0
axisP[0]	0	0	0	1	1
axisP[1]	0	0	0	1	1
axisP[2]	0	0	0	1	1
axisI[0]	0	0	0	1	1
axisI[1]	0	0	0	1	1
axisI[2]	0	0	0	1	1
axisD[0]	0	0	0	1	1
axisD[1]	0	0	0	1	1
axisD[2]	0	0	0	1	1
gyroADC[0]	0	0	0	3	1
gyroADC[1]	0	0	0	3	1
gyroADC[2]	0	0	0	3	1
accSmooth[0]	0	0	0	3	1
accSmooth[1]	0	0	0	3	1
accSmooth[2]	0	0	0	3	1
debug[0]	0	0	0	1	1
debug[1]	0	0	0	1	1
debug[2]	0	0	0	1	1
debug[3]	0	0	0	1	1
debug[4]	0	0	0	1	1
debug[5]	0	0	0	1	1
debug[6]	0	0	0	1	1
debug[7]	0	0	0	1	1
rcCommand[0]	0	0	8	1	1
rcCommand[1]	0	0	8	1	1
rcCommand[2]	0	0	8	1	1
rcCommand[3]	1	0	8	1	0
motor[0]	0	4	1	3	0
motor[1]	0	5	1	3	0
motor[2]	0	5	1	3	0
motor[3]	0	5	1	3	0

5.5 Pilot Controller Firmware

As *Pilot Controller* we choose the nRF52840. For the prototype quadcopter we use the Particle Xenon [Par] which is a development board around the nRF52840. Additional to the features of the nRF52840 described in Section 3.1.6 the board offers:

- $4MB$ SPI flash
- 20 mixed signal GPIO (6 x Analog, 8 x PWM), UART, I2C, SPI
- Micro USB 2.0 full speed (12 Mbps)
- JTAG (SWD) Connector
- Integrated Li-Po charging and battery connector
- On-board PCB antenna
- RGB status LED

Figure 5.16: Particle Xenon development board [Par].

The Particle Xenon is supported by the *Zephyr* project, a real-time operation system supported by the Linux Foundation [The]. An overview of the firmware is shown in Figure 5.17. Because we choose the *Decentralized Approach* in Section 3.1.4 the *Pilot Controller* implements the collision avoidance. The schematic of the quadcopter is shown in Figure 4.9. The source code for the firmware can be found here [Teme].

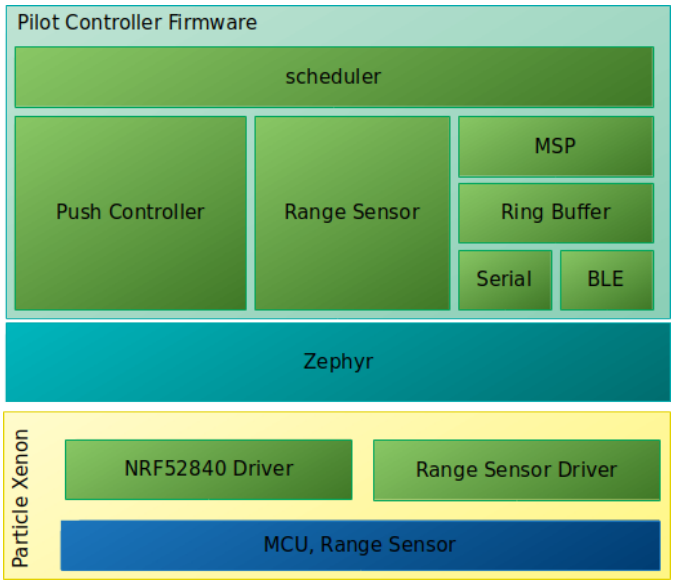


Figure 5.17: Firmware layers used by the *Pilot Controller*.

5.5.1 MSP

One major part of the *Pilot Controller* is exchanging of data. The *Pilot Controller* will decode each **incoming** MSP-frame. There are three options for further data processing, either consuming the frame, handing the frame over to the second interface or responding on the same interface on which the frame was received. A classification of the known frames for the *Pilot Controller* for each interface is shown in Table 5.4 and Table 5.3. All frames not listed are handed over to the corresponding second interface. Because MSP is a serial protocol and the speed of the two interfaces of the *Pilot Controller* differs we implement four ringbuffers (see Figure 5.18). These buffers hold the data and get decoded or are loaded with data by the scheduler in the main loop. The sending and receiving the data is done using interrupts.

On the PC we use *pygatt* to send and receive data via BLE. We implement the readout of the controller in python. As controller we support once more the *PS3 DualShok Controller* as well as the *Tanaris X7*. Because the communication via BLE is done by the MSP we implement this also in python. The main advantage of using also MSP over this interface is that we can now expose the interface also to the *Betaflight Configurator*. To do so we start a TCP server in python where the *Betaflight Configurator* can connect. Because of the limitation of 20 Bytes every 50ms the *Betaflight Configurator* acts slowly in that sense that for example the attitude preview is not smooth.

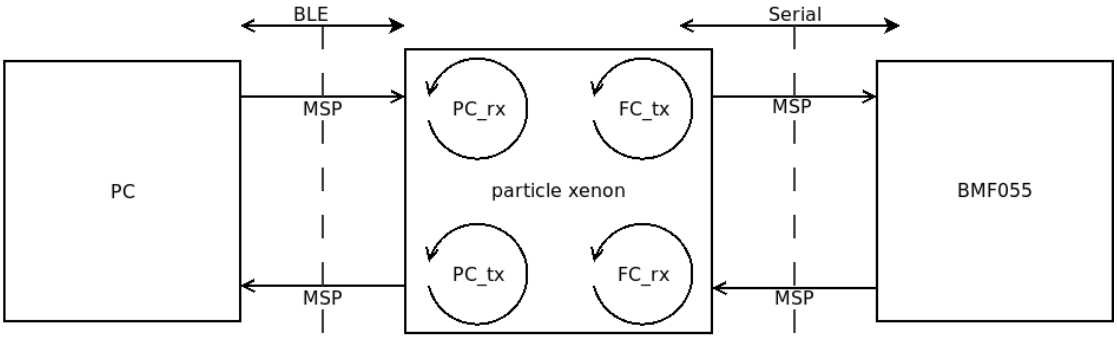


Figure 5.18: Data interfaces of the *Pilot Controller* with the associated ringbuffers to it.

Table 5.3: Classification of MSP-frames from the flight controller to the *Pilot Controller*.

command	code	frame type	decode type	description
MSP_ATTITUDE	108	response	CONSUME	includes 16-bit signed values (roll,pitch,yaw)
MSP_SET_RAW_RC	200	response	CONSUME	acknowledge from flight controller to request

Table 5.4: Classification of MSP-frames on the interface from the PC to the *Pilot Controller*.

command	code	frame type	decode type	description
MSP_ATTITUDE	108	request	REPLY	requests the estimated attitude of the quadcopter
MSP_SET_RAW_RC	200	command	CONSUME	includes the current steering information (rc-data)
MSP_SET_PID	202	command	REPLY	set the PID values of the collision and altitude hold controller
MSP_NAME	10	request	REPLY	get the name of the flight controller

5.5.2 Scheduler

The scheduler is a simple real time scheduler. The state machine executing is shown in Figure 5.19. The scheduler decides, based on the current task and the current time, if the next task is executed or skipped. The defined timings are:

- process MSP (every time)
- fetch sensor every **33ms**
- fetch attitude every **10ms**
- execute controller (every time)
- send RC Data every **20ms**

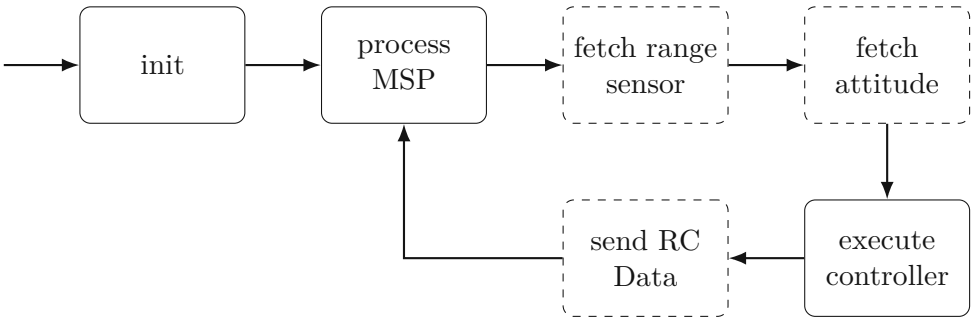


Figure 5.19: Scheduler used by the *Pilot Controller*.

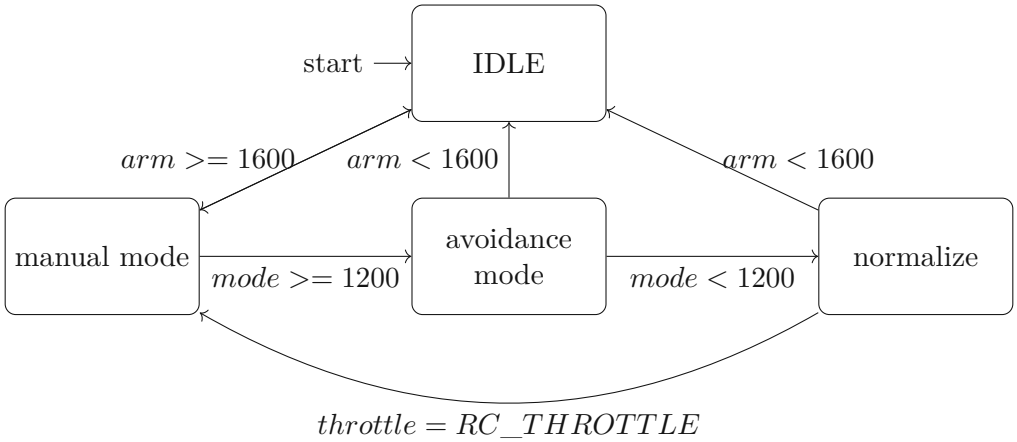


Figure 5.20: Flight mode state machine of *Pilot Controllers* main controller.

5.5.3 Main Controller

The *Pilot Controller* implements two operation modes.

- *manual mode*
  In this mode the quadcopter is controlled by the operator. The *Pilot Controller* passes the data unmodified to the flight controller.
- *avoidance mode*
  In this mode the quadcopter is executing the collision avoidance and altitude hold controller and therefore modifies the steering data sent to the flight controller.

Each time the **execute controller** method is scheduled a state machine for generating the output to controlling the flight controller is executed. Figure 5.20 shows a simplified version of the state machine. As long as $arm < 1600$ we stay in the idle mode. If $arm >= 1600$ we arm the quadcopter and enter the *manual mode* . If $mode >= 1200$ we activate the *avoidance mode*. If in *manual mode* or *avoidance mode* $arm < 1600$ we immediately enter the idle mode and unarm the quadcopter. The *normalize mode* is an intermediate mode where the throttle will slowly be increased/decreased to the throttle value of the operator. This prevents throttle jumps when disabling the *avoidance mode*.

5.5.4 Range Sensor

We use the *VL53L0X* as range sensor. The data we get from the VL53L0X contains noise therefore we filter the data. First the data is filtered by an exponential filter and then optionally filtered by a biquad filter. Figure 5.21 shows the output after the exponential filter. Figure 5.22 show the output if the optional biquad filter is active.

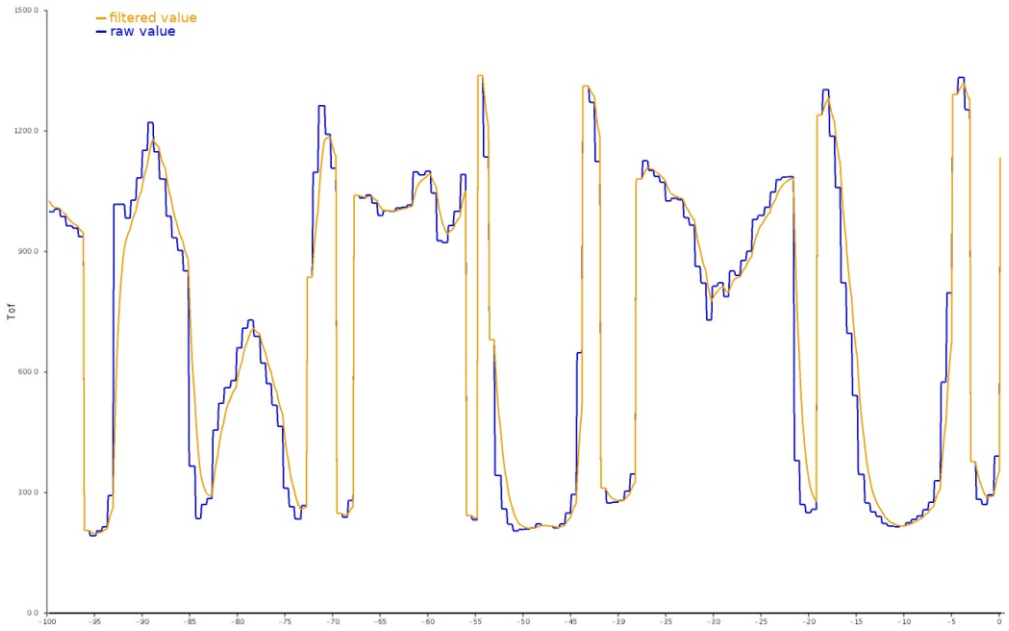


Figure 5.21: Rage sensor filtered by an exponential filter (x-axis = samples, y-axis = range in mm).

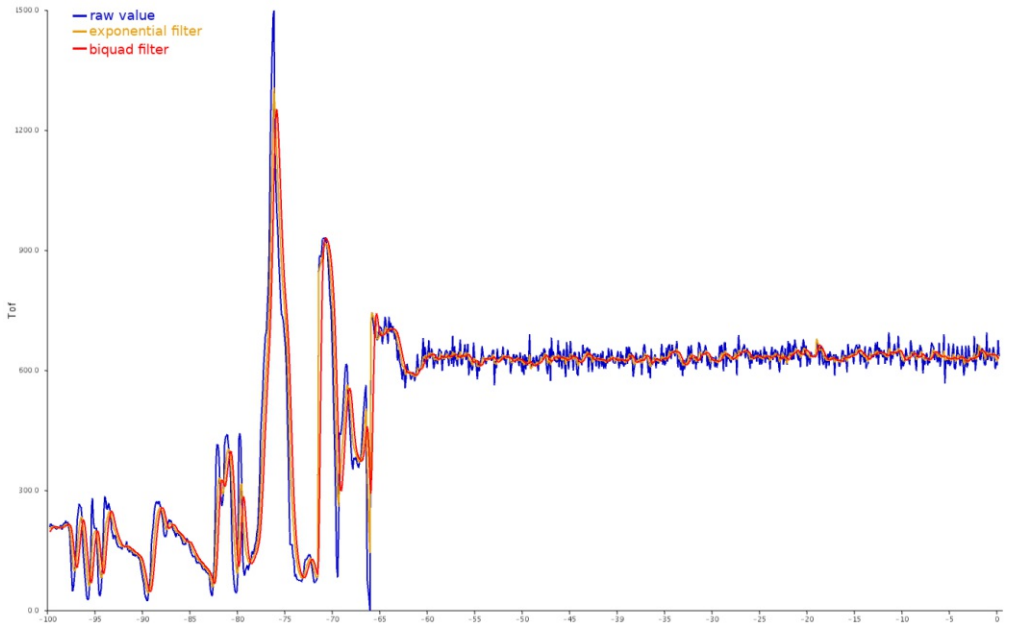


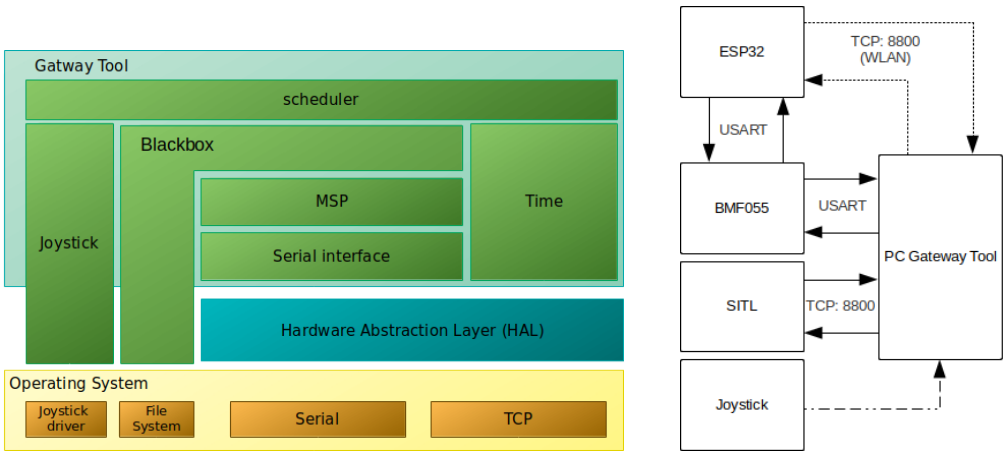
Figure 5.22: Rage sensor filtered by exponential filter and biquad filter (x-axis = samples, y-axis = range in mm).

5.6 PC Gateway Tool

For testing and debugging we develop a tool for communicating with the flight controller firmware. This tool has to support communicating with the actual hardware as well as with the SITL. We used the MSP protocol to communicate with the firmware. Fortunately, parts oft the firmware code is designed with this in mind so we can reuse components from the firmware here. Figure 5.23a shows an overview of the software design. Figure 5.23b shows an overview of the interfaces. For the *Serial Interface* a TCP and USART driver was written. With the TCP Port it is possible to connect the SITL or the *BMF055* (via the ESP32). This enables us to write the blackbox data received from the MSP to a file.

5.6.1 Esp32

Programming the ESP32 is done with the *Arduino IDE*. The ESP32 connects to a WLAN Router and creates a TCP socket where the *PC Gateway Tool* can connect to. It then buffers all the data received on the TCP socket and send it to the serial interface connected to the flight controller. All data received on the serial interface is sent to the TCP socket. We use a buffer for incoming characters on the serial interface because we don't want to send TCP packages with only a few bytes. This buffer waits $1ms$, for new data, or up to 1000 characters before transmitting.



(a) (b)

Figure 5.23: In (a) the software layers of the *PC Debug Tool* are shown, in (b) the defined interfaces for communication.

CHAPTER 6

Simulation and Tests

In this chapter we will first describe why a simulation makes sense for developing a quadcopter. We will then discuss existing simulation tools and which one we choose for the simulation. The first simulation setup is based on the *Betaflight Port Attempt*. The second simulation setup is based on the SITL feature of the flight controller firmware in Section 5.4. We will then describe how the prototype quadcopter versions, defined in Chapter 4, perform and how we tested them.

6.1 Simulation advantages and tools

To develop the collision avoidance a simulation environment was established first. A simulation has many advantages over the testing on the actual hardware. First we can avoid damaging the hardware on crashes. Second because the environment and quadcopter is simulated we get detailed debug information. And last but not least we do not need to load a battery for the flight which saves time and speeds up the development process.

The available simulation tools roughly divide in simulation for human to improve there skills and simulation for scientific usage like autonomous flight. The first kind of simulation was used to get familiar with quadcopters in a first place. The advantage of such a tool is the easy use and quick setup. To develop the collision avoidance, however, a scientific tool was needed.

This tool has to be easy to adapt, support laser range sensors and be as close as possible to the software implementation on the actual quadcopter.

To simulate a quadcopter we need a framework which can simulate the dynamics of a quadcopter and the world it is put in (physics engine). *Gazebo* [Ope] fulfills all this requirements but *gazebo* is only the base part. We also need a model of the quadcopter we can put into *gazebo*. This model includes the physical dynamics of the quadcopter including all sensors and the software to control it (flight controller). The first framework tested was the simulation provided by the PX4 flight controller. This simulation uses the PX4 flight controller as SITL [PX4b]. PX4 is an autopilot flight controller ecosystem with lots of features but for our simulation we only need the basic flight functionality. Another well tested and used framework is RotorS [FBAS16]. This framework is light-weighted and is easy to adapt to our needs. *RotorS* is a Micro Aerial Vehicles (MAV) simulation framework which uses *gazebo* and ROS. ROS is a collection of tools to simulate and program robots [QGS15].

6.2 *Betaflight Port Attempt* - Simulation Setup

The first simulation we have developed is for the *Betaflight Port Attempt* (see Section 5.3). The source code is available in [Temb]. The RotorS Framework defines multiple quadcopter models. For this simulation the asctec-hummingbird model was used. The simulation is based on the *Decentralized Approach* (3.1.3). In Figure 6.1 the ROS node graph is shown. Because the *Betaflight Port Attempt* firmware does not support SITL we implement the PID controller and motor mixer of the firmware in the *Flight Controller Node*. The *Flight Controller Node* does not implement an IMU. The state estimation was done by using the odometry sensor of the RotorS framework and is therefore optimal (without any errors). The interfacing between the *Pilot Controller Node* and *Flight Controller Node* was abstracted. This means we use ROS topics between the nodes instead of using the MSP. The *Pilot Controller* does not support SITL, therefore only necessary parts from the firmware are ported to the node. The node gets the necessary IMU information (attitude,..) directly from the simulation.

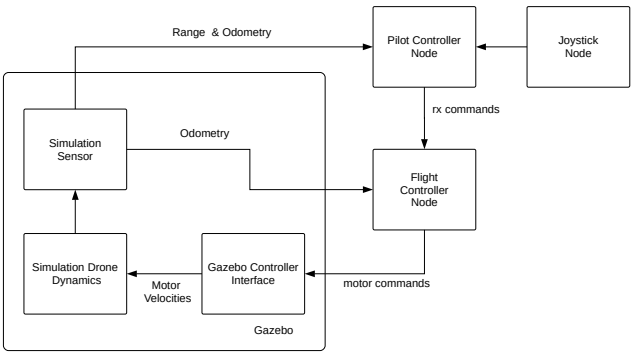


Figure 6.1: ROS Node Graph for *Betaflight Port Attempt* - Simulation Setup.

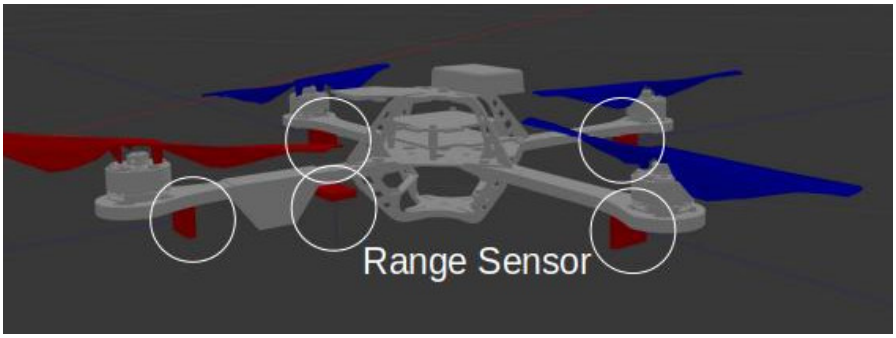


Figure 6.2: Range sensor mounted on asctec-hummingbird model.

To simulate the collision avoidance it was necessary to mount range sensors on the hummingbird model. To do so we first need to define the range sensor for the model. *Gazebo* supports ray sensors and we only need to define the sensor and configure it. The defined range sensor consists of a single laser ray with:

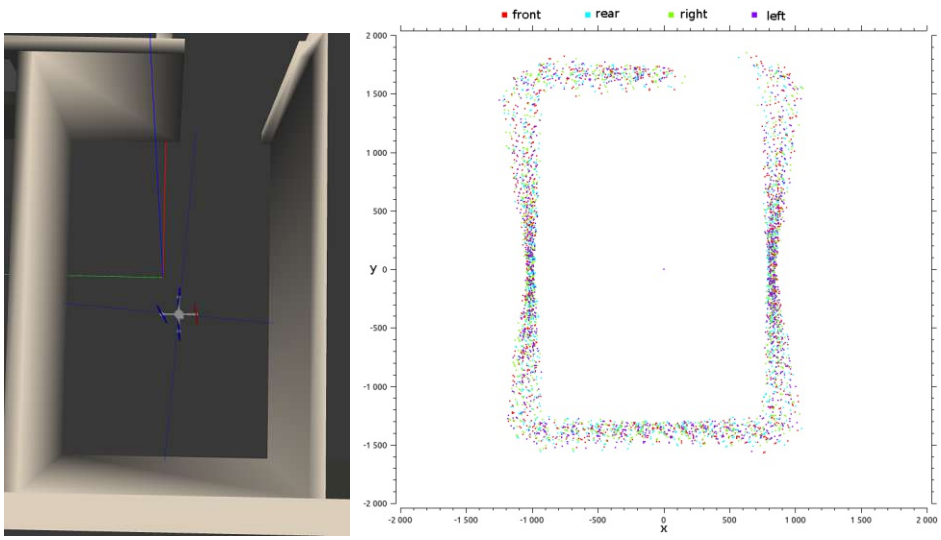
- minimum range of 2mm
- maximum range of 2m
- resolution 1mm
- gaussian noise with 1mm standard deviation
- new sensor value each $30ms$ (value of proposed sensor in 4.1)

We can now mount this sensor on the hummingbird model by overwriting the necessary *.gazebo* files in the launch process. We mount a sensor in each direction and one sensor downwards (see Figure 6.2).

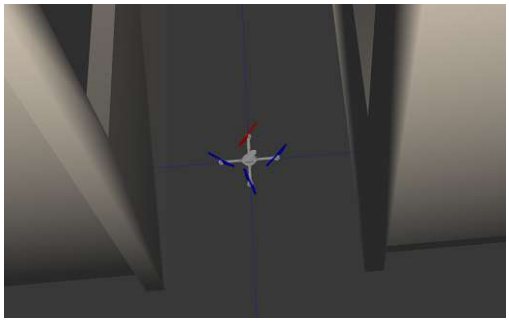
6.2.1 Results

Figure 6.3a and Figure 6.3b shows how the estimation of the quadcopter looks like if it rotates in place.

We now tested with the three different sensor arrangements as described in Section 3.2. With one front facing sensor the quadcopter was not always able to avoid objects because the main problem was the polling rate and the quadcopter flying in a direction were the sensor is not currently facing. With front facing sensor and back facing sensor this was less of a problem because we doubled the polling rate but now we encounter a new problem. If the quadcopter flies in a narrow passage (see Figure 6.3c) it can happen that the two *Push-Controllers* for front ant back rock up each other. This is solved by tuning the *Push-Controllers*. If we use sensors in all direction this problem gets less important because we once more double the polling rate. And so the *Collision Avoidance Controller* has it easier to correct an overreact of a *Push-Controllers*. With this sensor arrangement it was possible for the quadcopter to avoid most collisions in the simulated environment.



(a) (b)



(c) narrow passage

Figure 6.3: Environment estimation of quadcopter in simulation environment. In (a) the world the quadcopter is put in is shown. In (b) the estimation of the quadcopter is shown (values in mm). In (c) the a narrow passage is shown.

6.3 SITL Simulation Setup

After encounter problems during testing the collision avoidance with the *Betaflight Port Attempt* in real life we decided to also polish the simulation because the used simulation setup in Section 6.2 does not reflect the real life (too much abstraction). This time we use the flight controller firmware that supports SITL (see Section 5.4). This means we can run the control of the quadcopter with the code also used on the target hardware and do not need to extract this code (former *Baseflight Controller Node*).

6.3.1 Quadcopter Model

This time we replaced the asctec-hummingbird model with a more accurate model of the prototype quadcopter. To add a new model to the RotorS simulation framework we first build the meshes for displaying the quadcopter.

Meshes

This was done with help of the modeling tool *SketchUp* [Tri]. *SketchUp* is mainly for architecture purposes but it is one of the simplest 3d designing tools available. Important for our need is that *SketchUp* supports exporting to the *.dae* file format used by *gazebo* to load meshes. Because *SketchUp* is designed around architecture the tools are optimized for models in meter scale. So we decided to build the model ten times bigger then it actually is and scale it down later.

Fortunately, the *.dae* is simply an xml file containing all geometric figures and also an scaling factor for it. So we scale the whole model by just changing the *meter* attribute of the *unit* tag (see Listing A.1). The resulting model is shown in Figure 6.4.

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<COLLADA xmlns="http://www.collada.org/2005/11/COLLADASchema" version="1.4.1">
    <asset>
        <contributor>
            <authoring_tool>Google SketchUp 7.1.6860</authoring_tool>
        </contributor>
        <created>2020-08-07T15:13:29Z</created>
        <modified>2020-08-07T15:13:29Z</modified>
        <unit meter="0.000254" name="inch" />
        <up_axis>Z_UP</up_axis>
    </asset>
    <library_visual_scenes>
        ...
```

Listing 6.1: Quadcopter prototype model file: *bmflight_test_drone.dae*.

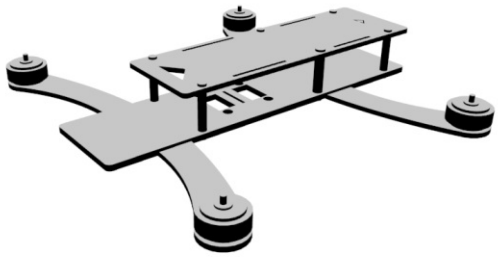


Figure 6.4: Quadcopter prototype model.

Flight Dynamics

The flight dynamics of the model are specified in one single *.xacro* file (see Listening 6.2) which will be loaded when the simulation spawns the model.

```
<?xml version="1.0"?>
<robot name="bmflight_test_drone" xmlns:xacro="http://ros.org/wiki/xacro">
  <xacro:property name="namespace" value="$(arg namespace)" />
  <xacro:property name="rotor_velocity_slowdown_sim" value="10" />
  <xacro:property name="use_mesh_file" value="true" />
  <xacro:property name="mesh_file" value="package://bmflight_simulation/meshes/
      bmflight_test_drone.dae" />
  <xacro:property name="mass" value="0.88" /> <!-- [kg] -->
  <xacro:property name="body_width" value="0.08" /> <!-- [m] -->
  <xacro:property name="body_height" value="0.06" /> <!-- [m] -->
  <xacro:property name="mass_rotor" value="0.005" /> <!-- [kg] -->

  <xacro:property name="rotor_offset_top" value="0.018" /> <!-- [m] -->
  <xacro:property name="arm_length_x" value="0.0848" /> <!-- [m] -->
  <xacro:property name="arm_length_y" value="0.1015" /> <!-- [m] -->
  <xacro:property name="radius_rotor" value="0.06" /> <!-- [m] -->
  <xacro:property name="motor_constant" value="8.54858e-06" /> <!-- [kg m/s^2] -->
  <xacro:property name="moment_constant" value="0.016" /> <!-- [m] -->
  <xacro:property name="time_constant_up" value="0.0125" /> <!-- [s] -->
  <xacro:property name="time_constant_down" value="0.025" /> <!-- [s] -->
  <xacro:property name="max_rot_velocity" value="838" /> <!-- [rad/s] -->
  <xacro:property name="rotor_drag_coefficient" value="8.06428e-05" />
  <xacro:property name="rolling_moment_coefficient" value="0.000001" />

  <!-- Property Blocks -->
  <xacro:property name="body_inertia">
   <!-- [kg m^2] [kg m^2] [kg m^2] [kg m^2] [kg m^2] [kg m^2] -->
    <inertia ixx="0.001438548933333" ixy="0.0" ixz="0.0" iyy="0.002462448810667" iyz="0.0"
        izz="0.003813904410667" />
  </xacro:property>

  <!-- inertia of a single rotor, assuming it is a cuboid. Height=5m, width=15mm -->
  <xacro:property name="rotor_inertia">
    <xacro:box_inertia x="${radius_rotor}" y="0.015" z="0.005" mass="${mass_rotor*
        rotor_velocity_slowdown_sim}" />
  </xacro:property>
        ...
```

Listing 6.2: Quadcopter prototype flight dynamics file *bmflight_test_drone.xacro*.

According to Förster [För15] getting accurate values for some of these parameters can be quite complicated. For example the mass, width and height can easy be measured, the *motor_constant* and *moment_constant* are quite hard to define. Therefore, we used the same motor/rotor model also used by the asctec-hummingbird model. We assumed in 2.1 that the quadcopter is symmetric in mass and geometry. Therefore from Luis [LN16] we

know the inertia matrix $\mathbf{J}$ is simplified to a diagonal matrix

$$\mathbf{J} = \begin{bmatrix} J_{xx} & 0 & 0 \\ 0 & J_{yy} & 0 \\ 0 & 0 & J_{zz} \end{bmatrix} . \tag{6.1}$$

From Tytler [Cha] we know we can compose the inertia of the quadcopter as a sum of inertias. We decided to split it up in the inertia of the four motors $\mathbf{J_M}$, the frame $\mathbf{J_F}$ and the center body $\mathbf{J_B}$.

$$\mathbf{J} = \mathbf{J_M} + \mathbf{J_F} + \mathbf{J_B} \tag{6.2}$$

We define the motor as point masses about d away from the center of mass, therefore, we can calculate the inertia of all 4 motors as sum of the point masses.

$$\mathbf{J_M} = \begin{bmatrix} J_{xx,M} & 0 & 0 \\ 0 & J_{yy,M} & 0 \\ 0 & 0 & J_{zz,M} \end{bmatrix} \tag{6.3}$$

$$J_{xx,M} = \sum_{i=1}^{4} m_M \left(y_m^2 + z_m^2\right) \tag{6.4}$$

$$J_{yy,M} = \sum_{i=1}^{4} m_M \left(x_m^2 + z_m^2\right) \tag{6.5}$$

$$J_{zz,M} = \sum_{i=1}^{4} m_M \left(x_m^2 + y_m^2\right) \tag{6.6}$$

Where m_M is the mass of a motor and x_m, y_m, z_m is the position of the point mass (see Figure 6.5a). The mass for the motors are equal and $z_m = 0$ so we can simplify Equations 6.4, 6.5 and 6.6 to Equation 6.7 and 6.8.

$$J_{xx,M} = 4\, m_M\, y_m^2 \quad , \quad J_{yy,M} = 4\, m_M\, x_m^2 \quad , \quad J_{zz,M} = 4\, m_M\, (x_m^2 + y_m^2) \tag{6.7}$$

$$\begin{aligned} \mathbf{J_M} &= 4\, m_M \begin{bmatrix} y_m^2 & 0 & 0 \\ 0 & x_m^2 & 0 \\ 0 & 0 & x_m^2 + y_m^2 \end{bmatrix} \\ &= \begin{bmatrix} 1.17 * 10^{-3} & 0 & 0 \\ 0 & 0.816 * 10^{-3} & 0 \\ 0 & 0 & 1.987 * 10^{-3} \end{bmatrix} \end{aligned} \tag{6.8}$$

Note all values of $\mathbf{J}$ are in $kg\, m^2$. $m_M = 0.0284kg\,, x_m = 0.0848m\,, y_m = 0.1015m$

For the center body $\mathbf{J_B}$ we choose a cuboid centered at the body axis (see Figure 6.5b) from Venugopalan [Sus] we know that

$$\begin{aligned}\mathbf{J_B} &= \frac{m_B}{12}\begin{bmatrix} y_B^2+z_B^2 & 0 & 0 \\ 0 & x_B^2+z_B^2 & 0 \\ 0 & 0 & x_B^2+y_B^2 \end{bmatrix} \\ &= \begin{bmatrix} 0.16*10^{-3} & 0 & 0 \\ 0 & 1.104*10^{-3} & 0 \\ 0 & 0 & 1.18*10^{-3} \end{bmatrix}\end{aligned} \tag{6.9}$$

$$m_B = 0.23kg + 0.3364kg = 0.5664kg$$

$$x_B = 0.15m$$

$$x_B = 0.15m$$

$$y_B = 0.05m$$

$$z_B = 0.03m$$

In Equation 6.9 m_B is composed of the weight of the battery and test stack (breadboard, sensors, PDB ,...). For the battery weight we used the average of both used batteries defined in 4.1. For the test stack approximate a weight of $0.3364kg$

For the frame we also choose a cuboid approximation centered at the body axis (see Figure 6.5c).

$$\begin{aligned}\mathbf{J_F} &= \frac{m_F}{12}\begin{bmatrix} y_F^2+z_F^2 & 0 & 0 \\ 0 & x_F^2+z_F^2 & 0 \\ 0 & 0 & x_F^2+y_F^2 \end{bmatrix} \\ &= \begin{bmatrix} 0.107*10^{-3} & 0 & 0 \\ 0 & 0.54*10^{-3} & 0 \\ 0 & 0 & 0,647*10^{-3} \end{bmatrix}\end{aligned} \tag{6.10}$$

$$m_F = 0.2kg$$

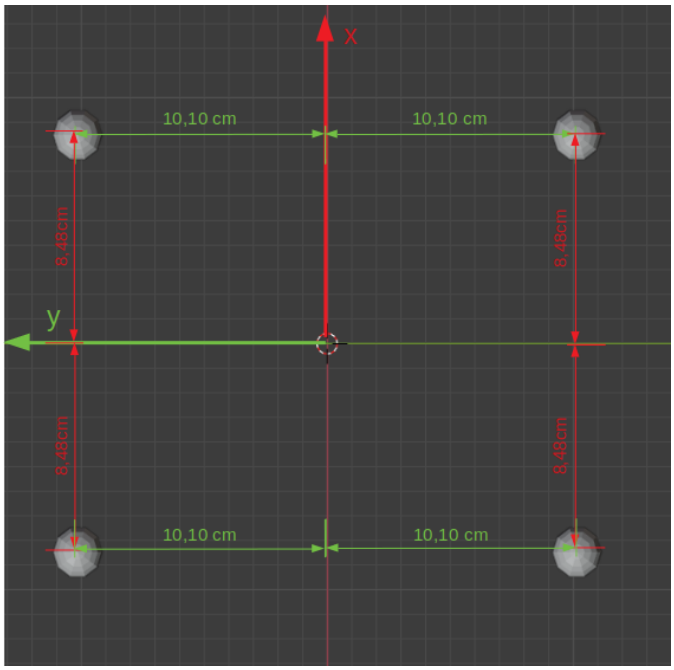
$$x_F = 0.18m$$

$$y_B = 0.08m$$

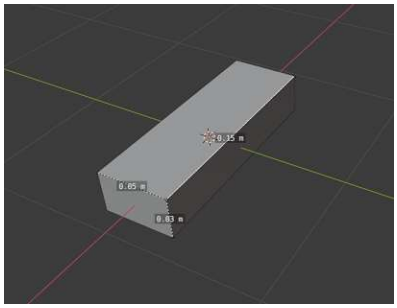
$$z_B = 0.008m$$

For the whole inertia matrix $\mathbf{J}$ we get then

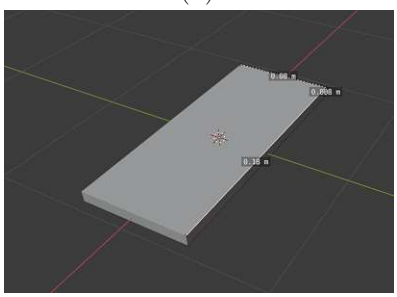
$$\mathbf{J} = \begin{bmatrix} 1.144*10^{-3} & 0 & 0 \\ 0 & 2.462*10^{-3} & 0 \\ 0 & 0 & 3.814*10^{-3} \end{bmatrix}. \tag{6.11}$$



(a)



(b)



(c)

Figure 6.5: Geometry for inertia calculation of quadcopter prototype. In (a) the geometry for the inertia $\mathbf{J_M}$ of the four motors are shown. In (b) the geometry for the battery inertia $\mathbf{J_B}$ is shown. In (c) the geometry for the inertia $\mathbf{J_F}$ of the prototype frame is shown.

6.3.2 SITL Integration

In Figure 6.6 the SITL integration for the flight controller firmware in the simulation environment is shown. The SITL (firmware) creates a TCP port where the *Gateway Tool* connects. The *Gateway Tool* is then used to sent controller data to the firmware and to store blackbox data received from the firmware (if enabled). For communicating with ROS we use the *SITL Interface*. This interface exchanges data with the firmware over User Datagram Protocol (UDP) ports. The *SITL Interface* is sending the current sensor data (accelerometer, gyroscope) and the current time. The firmware is sending the current output for the motors (PWM) and the estimated attitude.

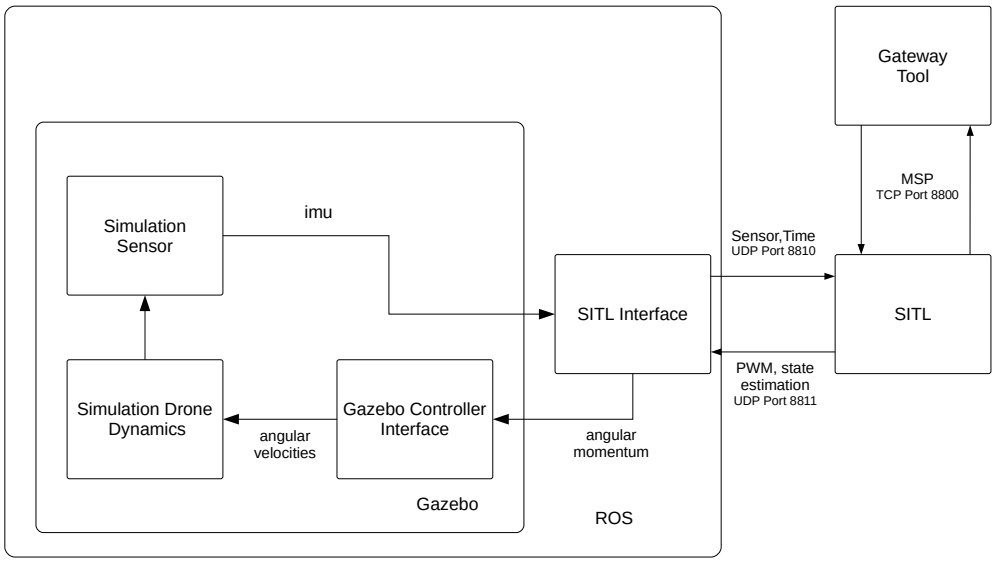


Figure 6.6: SITL integration for flight controller firmware.

The *SITL Interface* is responsible for converting the PWM values to angular momentum's for the motors. For the prototype this is done by the ESC. This controller aims to generate a linear thrust by controlling the power of the motor. From Figure 6.7 we see the motor almost generate linear thrust when controlled by power.

For the simulation we assumed that the angular velocity is quadratic

$$T_i = C_T \omega_i^2 \tag{6.12}$$

So we mapped the PWM signal p_i to angular velocity ω_i by

$$\omega_i = \sqrt{p_i - 1000} * k = \sqrt{p_i - 1000} * \left(\frac{\omega_{max}}{\sqrt{1000}} \right) \tag{6.13}$$

p_i is in the range of $[1000, 2000]$ and w_{max} is defined in the *bmflight_test_drone.xacro* by *max_rot_velocity* in 6.3.1.

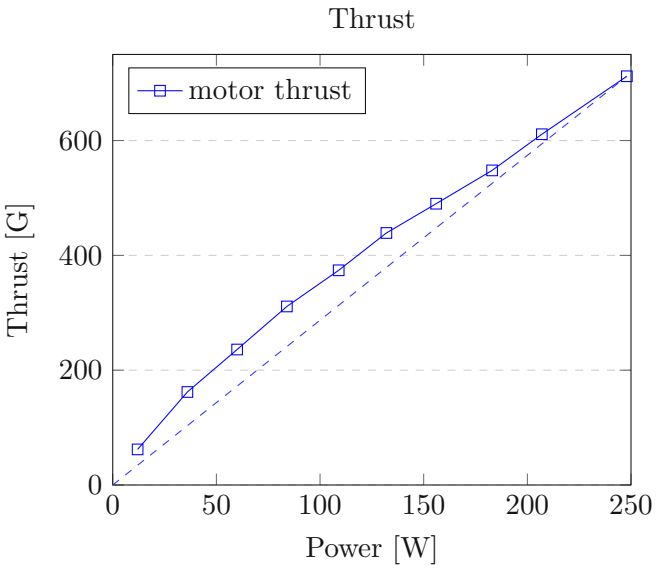


Figure 6.7: Thrust generated by ESC and propeller of prototype quadcopter.

In Figure 6.8 the integration of the *Decentralized Approach* in the simulation environment is shown. Because we do not have a SITL for the *Pilot Controller* we implement it as ROS node. The steering commands are sent via the *Joystick Node* to the *Pilot Controller Node*. The *Pilot Controller Node* sends the (manipulated) commands to the firmware by using the *SITL Interface* and *Gateway Tool*. The *Pilot Controller Node* gets the range values over ROS messages.

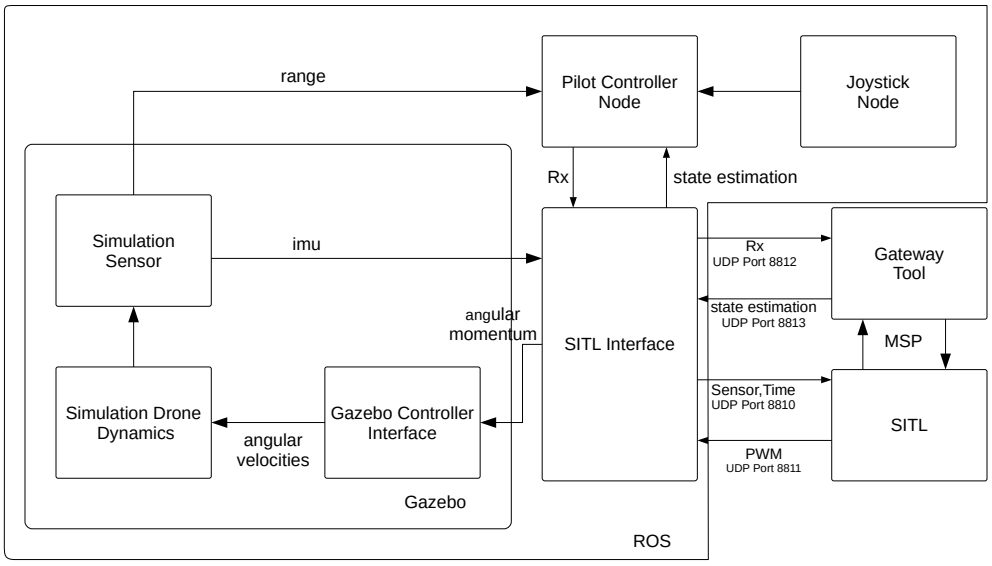


Figure 6.8: SITL integration for *Decentralized Approach*.

6.3.3 Results

For tuning the PID controller we use the blackbox module to measure the internals of the quadcopter during simulation. This shows the advantage of the SITL because such a tuning process was not possible with the firmware defined in Section 5.3.

Rate-Controller

It was already possible to fly the quadcopter with only a proportion controller (due to its highly symmetric design). In Figure 6.9 $kp = 60$ like expected we encounter settling process and persistent deviation. If we decrease $kp = 40$ (see Figure 6.10) this effect is less noticeable but we need longer to reach the desired angular rate.

If we set the $kd = 100$ we get rid of the oscillation at the settling process which is shown in Figure 6.11. The yellow line shows how the derivation term counteracts the oscillation.

To test the integral term we mounted a sensor with $50g$ weight at the tip of the quadcopter so the quadcopter started to drift without the term. In Figure 6.12 all terms are used.

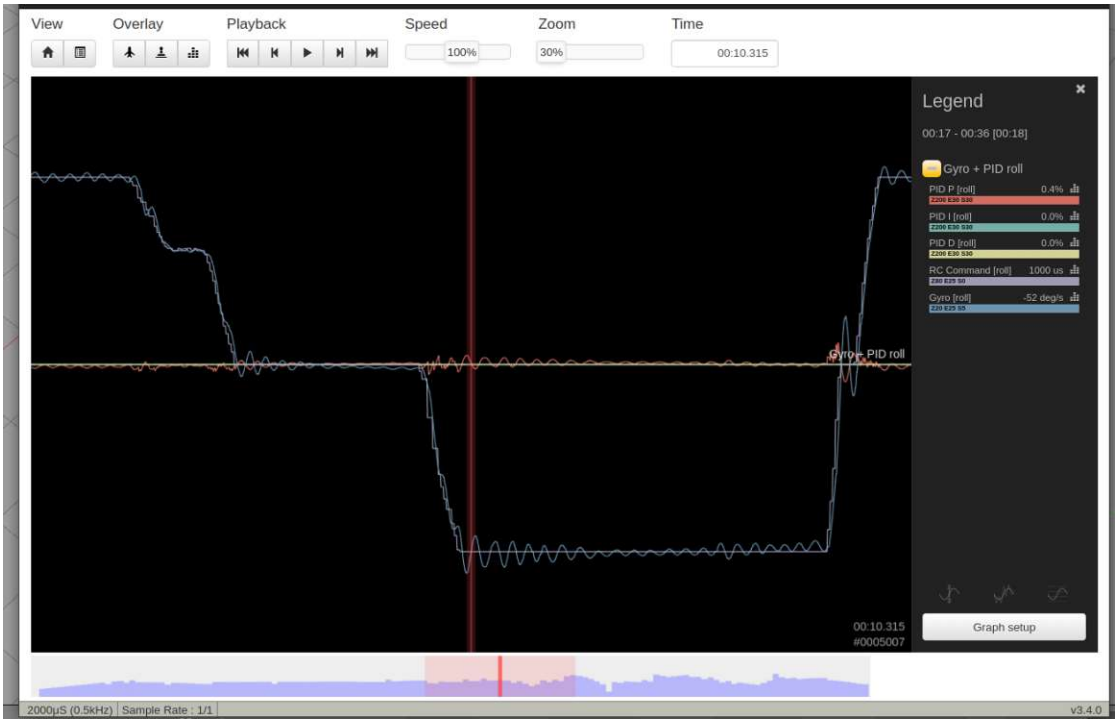


Figure 6.9: *Rate Controller* test for $kp = 60, ki = 0, kd = 0$.

Figure 6.10: *Rate Controller* test for $kp = 40, ki = 0, kd = 0$.

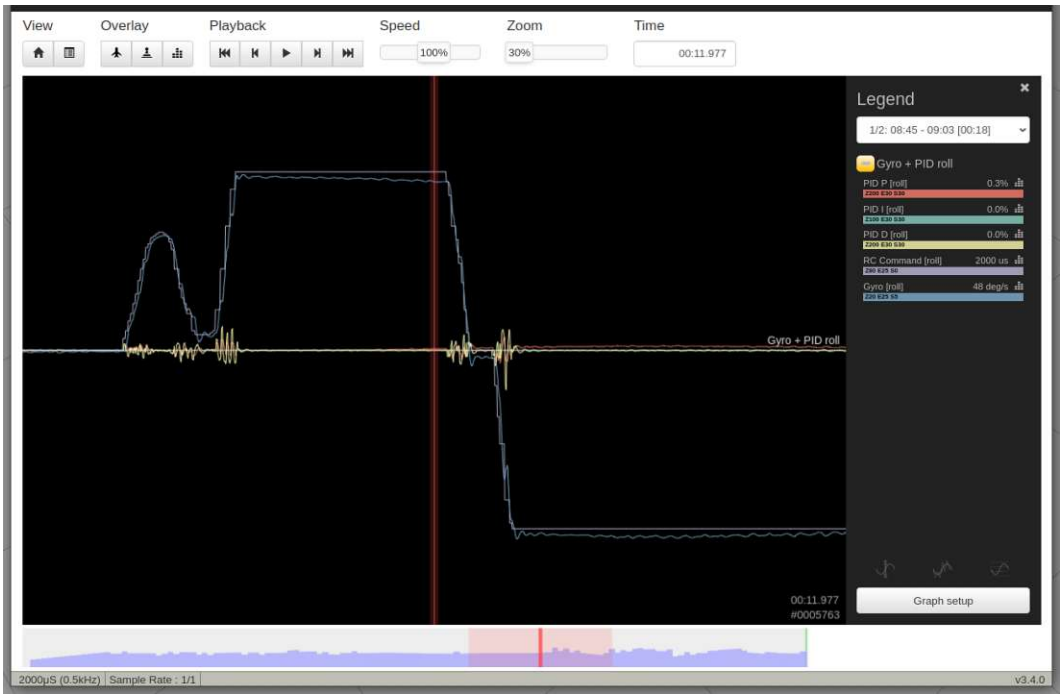


Figure 6.11: *Rate Controller* test for $kp = 60, ki = 0, kd = 100$.

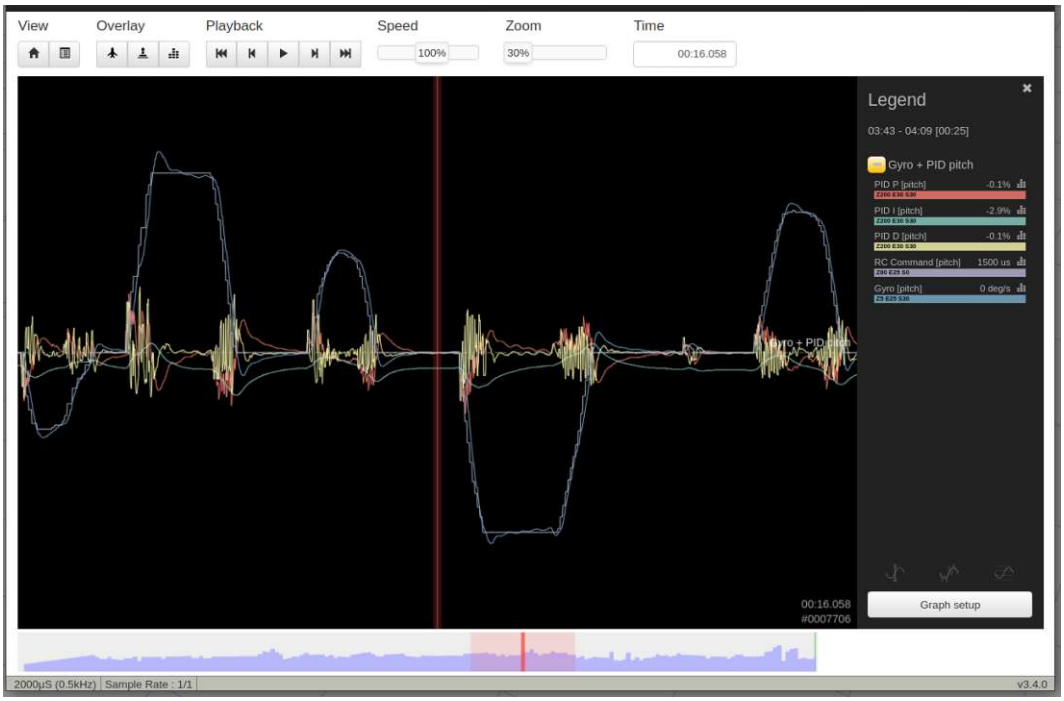


Figure 6.12: *Rate Controller* test for $kp = 60, ki = 40, kd = 100$. *PID I [pitch]* shows how the integration term is regulating the asymmetry from the mounted sensor.

6.4 Tests

While simulation can speed up the development time it can not completely replace testing on the actual prototype hardware.

6.4.1 Test for Altitude Hold Build

For this build there are problems encountered with the *Simblee* Module. It often loses the connection from the PC to the module. After quite some debugging time we could not find the problem. One major issue is as simple as it is to program with the *Arduino IDE* it hides quite away some complex mechanics. One guess was that the baudrate of the serial interface to the flight controller was too high. We used a baudrate of $115200Bd$ to get compatibility to the *Spektrum 2048* protocol but this was only possible bye overriding the UART limiter of the *simblee* library. In the library the UART was limited to $9600Bd$. So to test if this is the problem we limited the UART back to $9600Bd$. But unfortunately this does not solve the random connection loses. The altitude PID controller could be tuned so that the quadcopter was holding its altitude. This was done by take off the quadcopter in manual mode and then activate the hold mode. The response of the control was slow to range changes (for example an object under the quadcopter) as well as some oscillation was encounter if the change was too big. This has mainly two reasons first the control loop can take more than 150ms and second we don't have an exact correlation from the sensor value to time. The range value feeds to the PID could be 50ms or more than 150ms old.

6.4.2 Tests with Betaflight Port Attempt

In this test we used the prototype in his final build (see Section 4.5). The firmware used on the *BMF055* is the *Betaflight Port Attempt*. The quadcopter was able to hold its altitude. Also the *Head Free* mode worked well. It was also possible to avoid a collision if the quadcopter does not rotate. As soon as the quadcopter starts rotating, it became unstable if it detects an obstacle and tries to avoid it. It was hard to debug where this instability came from because we do not get any debug data in flight from the quadcopter. We assume that one reason is that the range sensor signal gets very noise and second the IMU of the *Betaflight Port Attempt* has shifted too strong. So we decided to build a test bench for the IMU to confirm this.

6.4.3 Inertial Measurement Unit Tests

Testing the IMU is challenging, simple tests by rotating the flight controller per hand are easy but inaccurate. So we decided to develop a test bench for that. The attitude of the quadcopter includes roll, pitch and yaw. We decided to test the yaw by the test bench, because it is mechanically challenging to build a test bench for all three dimensions. Moreover the yaw is a difficult part for an IMU to track because the accelerometer can not help in the sensor fusion algorithm (as described in Section 2.2).

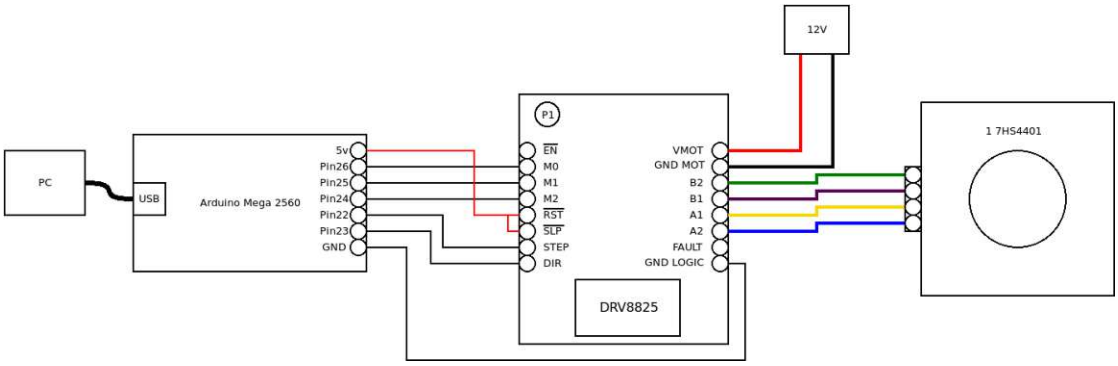


Figure 6.13: Schematic of IMU test bench.

Test Bench Hardware

To rotate the flight controller we used a stepper motor. The stepper motor used is the *7HS4401 (NEMA 17)* with a base resolution of 200 steps per revolution and an enough holding torch to carry the test breadboard and a battery. The stepper motor is controlled by the *DRV8825* stepper motor driver. This driver supports micro stepping so we can further increase the resolution of the motor. For controlling the driver we used an *Arduino Mega 2560*. To rotate the motor we only need to send a rising edge on the *STEP* line of the motor driver. A schematic of the test bench is shown in 6.13.

Test Bench Software

The software is split on different hardware (see Figure 6.14).

- PC controlling and logging the test
- *Arduino Mega 2560* controlling the stepper motor
- *Pilot Controller* node and the flight controller (device under test)

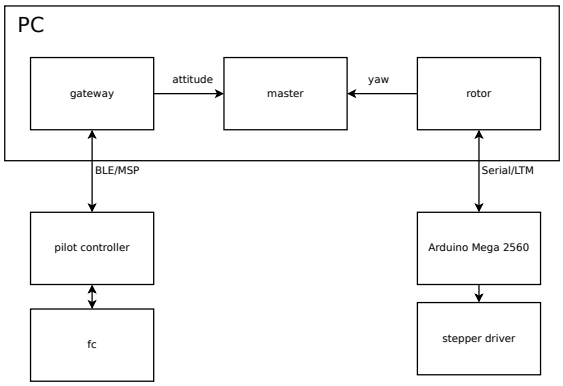


Figure 6.14: IMU test bench software components.

The *Pilot Controller* and flight controller don't need any adjustments because we can get the current attitude by sending an $MSP\_ATTITUDE$ request to the *Pilot Controller* (as described in Section 5.5.1). On the PC we have three ROS nodes. A *master node* for logging and synchronizing data, the *gateway node* for requesting data from the *Pilot Controller* and the *rotor node* which is controlling the *Arduino Mega 2560*. The stepper is controlled by the *rotor node* via the *Arduino Mega 2560*. We decided to use the LTMv2 protocol over the serial interface of the *Arduino* for the necessary communication. We defined the following frames:

- **U-Frame**: [$][*T*][*U*][*id*][*steps*][*speed*]
  - *id*...step command id (2 bytes)
  - *step*...absolute steps (4 bytes)
  - *speed*...speed to rotate to absolute steps (2 bytes)
- **S-Frame**: [$][*T*]*S*][*id*][*steps*][*speed*][*currentSteps*]
  - *id*...step command id (2 bytes)
  - *step*...absolute steps (4 bytes)
  - *speed*...speed to rotate to absolute steps (2 bytes)
  - *currentSteps*...current stepper steps (4 bytes)
- **R-Frame**: [$][*T*][*R*]

  reset command
- **E-Frame**: [$][*T*][*E*]

  error frame command
- **F-Frame**: [$][*T*][*F*][*id*]
  - *id*...step command id (2 bytes)

The software on the *Arduino Mega* is kept simply because timing is critical to achieve a certain number of revolution. Figure 6.15 shows the state machine implemented, blue indicates incoming frames and red outgoing frames. The *Arduino* waits after boot for the first U-Frame. The U-Frame has the steps the rotor is rotating clockwise and how fast it should rotate. We always track the absolute steps in clockwise direction. So if we want to rotate counter clockwise we first rotate clockwise for example step 16000 and then rotate to step 1600. Because we have 1600 steps per convolution this will mean ten rotation clockwise and then nine rotation counter clockwise. It is also possible to pause the rotation by setting the speed in the U-Frame to zero and define in the step data field the milliseconds to wait. In the *rotor node* we hold a list of rotation and speed we want to test. Each time the *Arduino* indicates the finishing of a rotation we start the next one by sending an U-Frame to the *Arduino*.

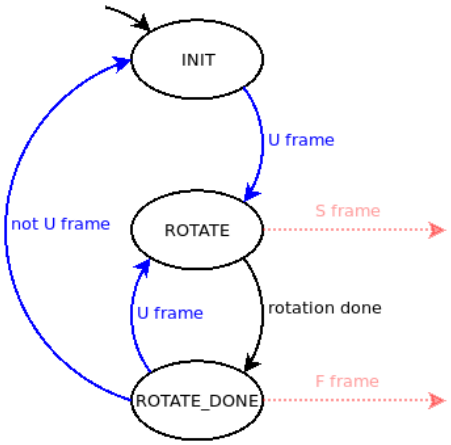


Figure 6.15: State machine executed by rotation controller (*Arduino Mega 2560*).

Test results for *Betaflight Port Attempt*

This test was done with the *Betaflight Port Attempt* and no active magnetometer. Figure 6.16 shows the results for the first test pattern (see Table 6.1). On the left axis the yaw is displayed on the right axis the error is displayed. On the top the id of the current active step in the pattern is stated.

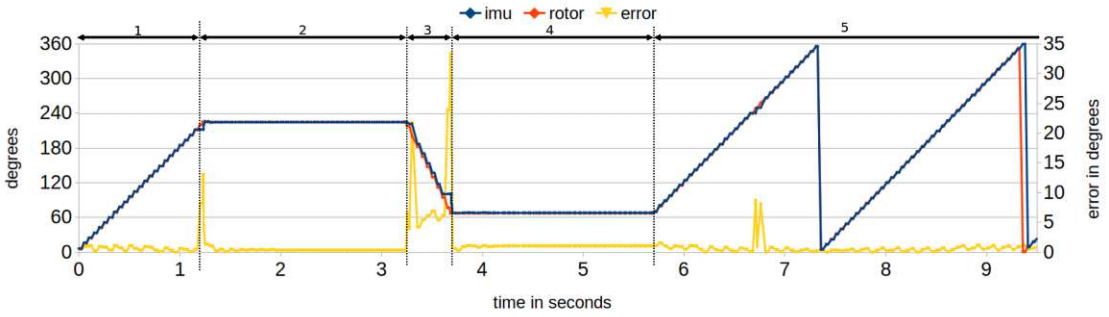
Table 6.1: Test pattern 1 for IMU test.

id	steps	degree	degree/second	id	steps	degree	degree/second
1	1000	225	180	7	3200	0	180
2	2000	225	0	8	1600	0	360
3	300	67,5	360	9	1000	225	540
4	2000	67,5	0	10	400	90	360
5	4800	0	180	11	20	4,5	180
6	2000	0	0	12	16000	0	180

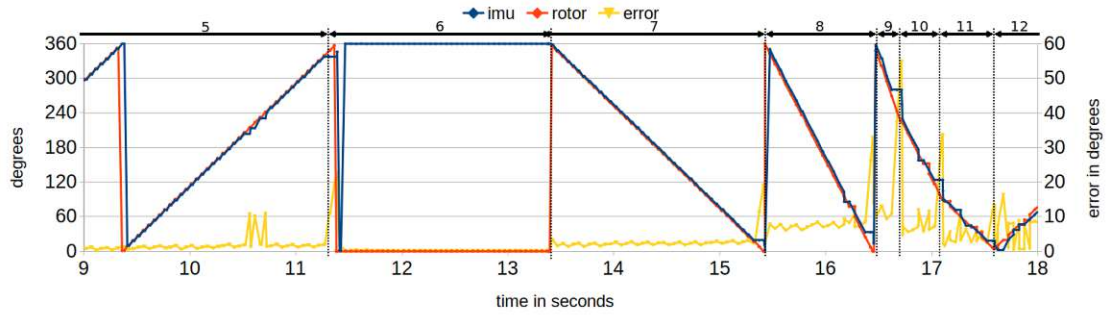
Table 6.2: Test pattern 2 for IMU test.

id	steps	degree	degree/second
1	16000	0	180
2	0	0	180
3	32000	0	180

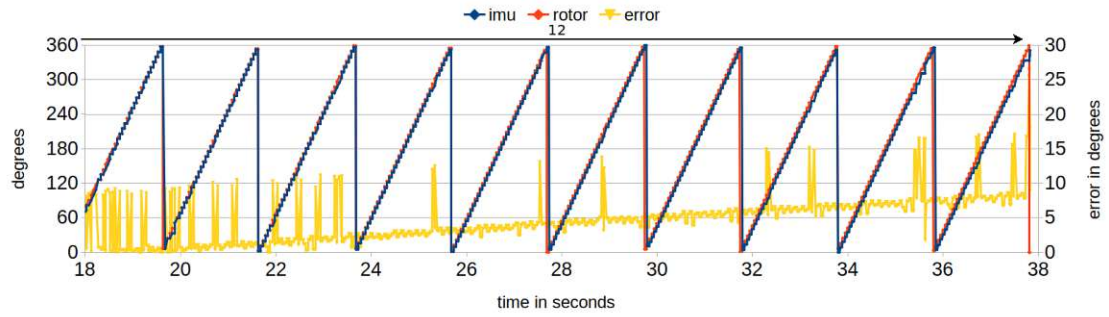
The IMU can follow the first eleven ids (see Figure 6.16a) with a mean error of about 2 degrees. Sometimes the errors spikes for example at about one second. The reason can be that the BLE connection was not able to deliver a new value in time so the *master node* has to take the old value. Or there was an abrupt change in velocity and due to the mechanical construction of the test bench we encounter a sway on the test subject. In Figure 6.16b we can also see this behavior on faster rotation the mean error is 6 degrees. In Figure 6.16c we can see a drift of the yaw on constant rotation.



(a)

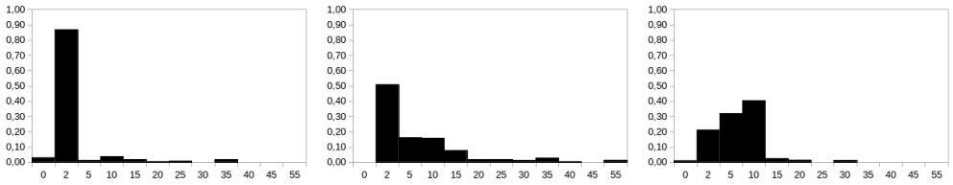


(b)



(c)

Figure 6.16: IMU test results for test pattern 1. (a) shows the seconds 0 to 9, (b) shows the seconds 9 to 18 and (c) the seconds 18 to 38.

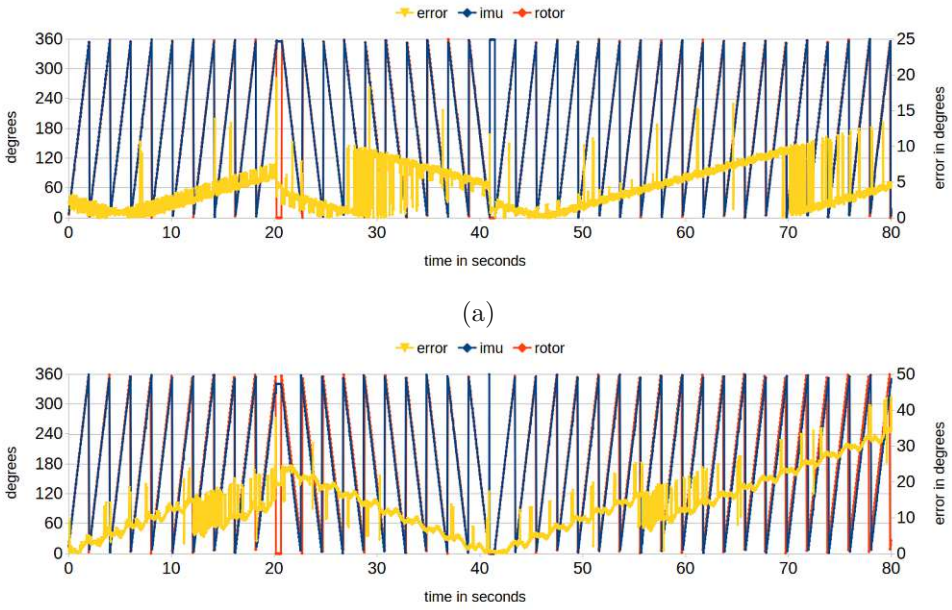


(a) (b) (c)

Figure 6.17: Histogram for error of test pattern 1. (a) shows the seconds 0 to 9, (b) shows the seconds 9 to 18 and (c) the seconds 18 to 38.

The second test was done with test pattern shown in Table 6.2. Here we longer rotate in one direction to test the drift we encountered in test pattern 1. The first 20 seconds we rotate clockwise, then 20 seconds counter clockwise and then 40 seconds clockwise. In Figure 6.18a we can see that the error increases until direction change. We also tested this with a pitch of 15 degrees, the results can be seen in Figure 6.18b.

The drift of the yaw we encounter has mainly two reasons, first the gyroscope inaccuracy (no magnetometer, therefore no correction of yaw drift) and second the test was mistakenly done with a high *"dcm_kp"* (ten times) of the IMU, because the quadcopter was not armed, therefore too much of the accelerometer is used which results in worse error on a pitch of 15 degrees.



(a)

(b)

Figure 6.18: IMU test results for test pattern 2. (a) shows results without pitch, (b) shows results with 15 degrees pitch.

6.4.4 Scheduler Timing Tests

In Figure 6.19 you can see the problems resulting from the dynamic scheduler strategy. The resulting period time for the tasks *TASK_LOOP* and *TASK_ATTITUDE* is very jittery as it often spikes up to 2.2ms. For the mean period time for *TASK_LOOP* we get 2019μs and for *TASK_ATTITUDE* 4040μs. In contrast if we use the static scheduler strategy the resulting period time is nearly constant with a mean period time of 2004μs for *TASK_LOOP* and 4007μs for *TASK_ATTITUDE* see Figure 6.20.

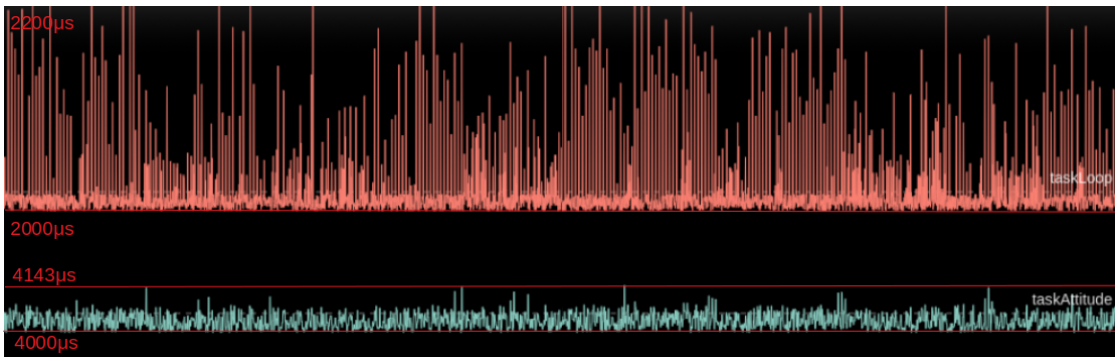


Figure 6.19: Period timing for *Dynamic Scheduler Strategy*.

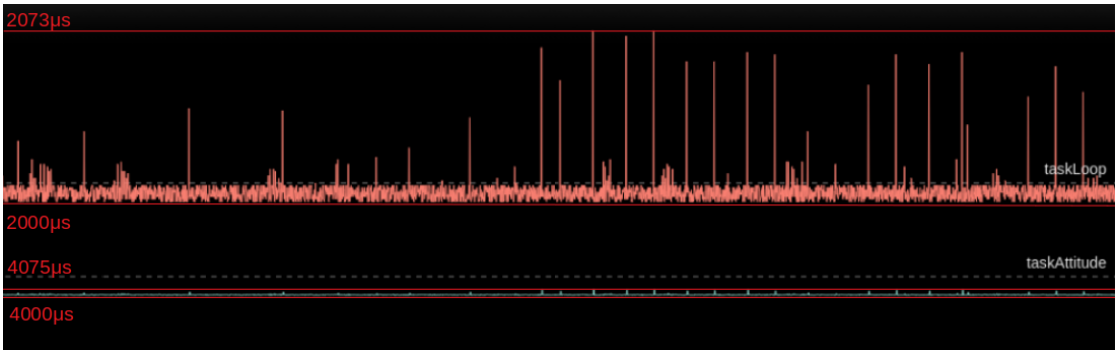


Figure 6.20: Period timing for *Static Scheduler Strategy*

CHAPTER 7

Conclusion

Aim of this work was to develop a collision avoidance for a quadcopter based on a front facing lidar sensor using the *BMF055* as flight controller. We used a decentralized system design where the tasks are split between the *Pilot Controller* and the flight controller based on the *BMF055*. This was done because of the limitation of the *BMF055*. With the first flight controller firmware (*Betaflight Port Attempt* see Section 5.3) we tried to port *Betaflight* to the *BMF055*. During development we realized that this was too ambitious because we needed many adaptation to the code base. Therefore, we take the missing parts from Blocher's work [Blo] so we can start developing the *Pilot Controller* and collision avoidance. For developing the collision avoidance we introduced a simulation in Section 6.2. This simulation used optimal versions of the IMU and range sensor so they have no noise or distortion. Even with these assumptions it was not possible to avoid all objects with only one front facing sensor. This was due to the low sampling rate and continuous rotation of the quadcopter (flying in a direction where the sensor is not currently facing). If we mounted four sensors in all directions of the quadcopter, the quadcopter was able to avoid most collisions in the simulation. But with the rotation the quadcopter is harder to control so the benefit over no rotation is not enough to justify the rotation in our opinion. A video from the simulation is available in [Temc]. Nevertheless we build a prototype quadcopter (Section 4.5) to find out if these results also translate into the real world. With the *Betaflight Port Attempt* firmware the quadcopter was able to avoid collisions without rotations. If the quadcopter starts to rotate the quadcopter gets unstable if it detects an obstacle. For better debugging we decided to rewrite the firmware of the *BMF055*. This time we aim for better debugging and hardware abstraction to make a SITL build of the firmware possible. The advantage of the proposed firmware is even though focused on the basic flight operation it also supports features *Betaflight Configurator* and *Betaflight Blackbox*.

With this new firmware we build a second simulation setup in Section 6.3 where we take use of the SITL feature. This simulation setup does not implement the collision avoidance at this point. The *Pilot Controller Node* implements the altitude hold controller but not the *Head Free Mode* and *Collision Avoidance Controller* proposed in Section 3.2. In Section 6.3.3 we nevertheless can already show the advantages over the first firmware implementation by using the blackbox module for tuning the PID controller for the simulation.

7.1 Further Task and Outlook

Further work can build up on the firmware and simulation supporting SITL by first improving the simulation by implementing the *Head Free Mode* and *Collision Avoidance Controller* parts in the *Pilot Controller Node* (can be copied from the node implemented in Section 6.2). The communication to the SITL can be improved by implementing the MSP in the *SITL Interface* so the *Gateway Tool* is no longer needed. This also enables reading the blackbox data in the *SITL Interface*. Here the frames can be decoded in real time and published to the ROS environment for real time analysis. We also can use the debug data to analyze and optimize the IMU drifting problems we encounter in Section 6.4.3. As described in Section 2.2 one step to takle the drifting problem is to add a magnetometer to the IMU.

To improve the collision avoidance a global position system can be established. With such a system the quadcopter knows where it is in the inertial frame and it can build a prediction of the environment from the sensors. Here the rotation approach of the quadcopter can show its full potential. Because it generates more data than without rotation so the estimation of the environment is more accurate.

After improving the simulation and collision avoidance, flight tests on the prototype quadcopter can be done using the blackbox module to get the internals of the quadcopter and tune the controllers based on this data. As *Pilot Controller* we choose the nRF52840 which is a solid foundation for future works, like autonomous flight and communication with sensor swarm.

APPENDIX A

Appendix

A.1 Particle Xenon Range Sensor Driver

The *VL53L0X* supports a I2C interface for API calls. *Zephyr* provides a driver but unfortunately, the driver, only supports one sensor on one I2C interface. Because we use two sensors at the same interface we adapt the driver. The I2C address for the *VL53L0X* is predefined by the sensor and needs to be changed on sensor initialization to support multiple sensors on the same interface. To make this possible the *VL53L0X* provides a *xshut* pin. If this pin is "low" the controller will not listen to the I2C interface. The driver therefore executes on boot the following tasks:

- initialize both *xshut* pins to output "low"
- activate sensor one and change I2C address
- setup sensor one (range profile)
- activate sensor two and change I2C address
- setup sensor two (range profile)

The new addresses of the sensor can be defined in the *particle_xenon.overlay*:

```
1  &i2c0 { /* feather I2C */
2          status = "okay";
3          clock-frequency = <I2C_BITRATE_FAST>;
4          sda-pin = <26>;
5          scl-pin = <27>;
6
7      vl53l0x@30 {
8                  compatible = "st,vl53l0x";
9                  reg = <0x30>;
10                 label = "VL53L0X_0";
11                 xshut-gpios = <&gpio0 3 GPIO_ACTIVE_HIGH>;
12         };
```

```
13          vl53l0x@31 {
14                  compatible = "st,vl53l0x";
15                  reg = <0x31>;
16                  label = "VL53L0X_1";
17                  xshut-gpios = <&gpio0 4 GPIO_ACTIVE_HIGH>;
18          };
19  };
```

Listing A.1: particle_xenon.overlay

A.2 BMF055 Programming

As IDE we used *eclipse*, to program the controller we used the *IBDAP*. The *IBDAP* is a *CMSIS-DAP JTAG/SWD* debug adapter for programming and debugging ARM Cortex MCUs [Arm]. We used the Serial Wire Debug (SWD) interface to connect the *IBDAP* to the BMF055 see Figure A.1. On the PC we used OpenOCD [Rat] as debugging software because it is well supported by the eclipse IDE.

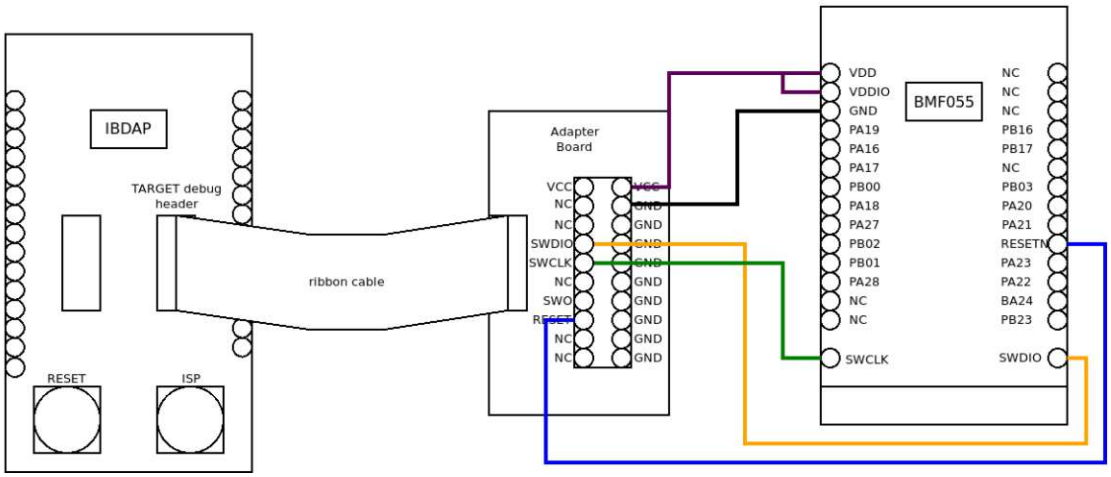


Figure A.1: Sketch for connecting the *IBDAP* programmer to the BMF055.

A.3 Flight Controller Firmware - BMF055 Driver

The drivers for the firmware are based on the Atmel Software Framework (ASF). The ASF is an abstraction for interacting with the hardware of the *samd20j18a*. Because we don't use Atmel Studio as IDE we have to import the necessary files by hand from the ASF. Blocher's work in [Blo] and the *Data Stream example* (Figure A.2) from Bosh [Bos] was here helpful to find the necessary parts and get them configured. We will now discuss the configuration and driver based on it.

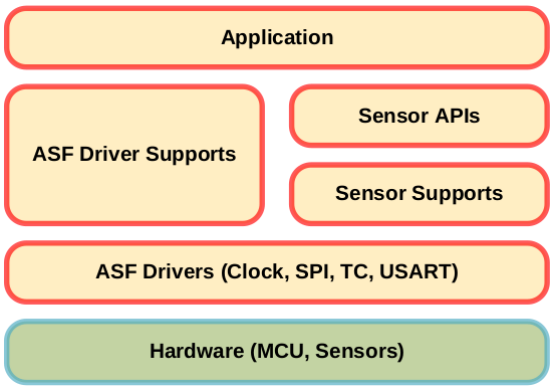


Figure A.2: Design structure of the BMF055 Data Stream [Bos] example.

A.3.1 Configuration

- **Clock**
  The *samd20j18a* used by *BMF055* supports multiple clocks to drive the hardware modules. For the main clock (*GCLK_GENERATOR_0*) we use the 48MHz Digital frequency locked loop (DFLL), DFLL48M as clock source. For the system time base we define the *GCLK_GENERATOR_1*. This Generic Clock uses the undivided 8MHz internal oscillator (OSC8M) as clock source and divides it by 8. For the USART we define the *GCLK_GENERATOR_2*. It also uses the OSC8M as clock source this time without a division factor. For the motor PWM signal we define the *GCLK_GENERATOR_3*. It also uses the OSC8M as clock source without a division factor.

- **SPI**
  The sensors of the **BMF055** are connected to the *samd20j18a* via SPI. As clock source we use the system clock *GCLK_GENERATOR_0*. The baudrate is set to $10Mhz$.

- **Electrically Erasable Programmable Read-only Memory (EEPROM)**
  The *samd20j18a* does not have an EEPROM therefore the ASF provides an emulation, done with the Non-volatile memory (NVM) on the *samd20j18a*. NVM is organized into rows (see Figure A.3) each of these rows containing four pages. A page contains *64 Bytes* of data. Deleting data can only be done by deleting the whole row. Writing can be done on page granularity. The emulation then abstracts this management of writing and reading data away. To enable the emulation first the NVMCTRL_EEPROM_SIZE fuse needs to be set.

Page $(n * 4) + 3$	Page $(n * 4) + 2$	Page $(n * 4) + 1$	Page $(n * 4) + 0$

Figure A.3: Row layout of NVM.

- **Timer/Counter**
  The *samd20j18a* supports up to eight 16-bit Timer/Counter (TC). To implement a 32-bit counter the *samd20j18a* pairs the even numbered TC with the following odd numbered TC.

  For the system time base we define the TC0 as 32-bit TC this means it is paired with the TC1. As clock source we use the *GCLK_GENERATOR_1*. This clock is 1MHz so the timer is increased every $1\mu s$. The number of microseconds since boot is now a readout of the 32-bit register of TC0. This means the TC will overflow at about 71 minutes. This is not critical since we use this value mainly for the scheduler and an overflow after 71 minutes results in a delay of 2ms for the main loop.

  For the motor driver we use two TC. This TC instance cannot be chosen freely because we need the waveform output (WO). The reason why we only need two TC and not four is that the *samd20j18a* supports two compare values (also called compare channels (CC)) for one TC. We use TC4 where CC0 is routed to WO0 (motor 1) and CC1 is routed to W01 (motor 2). As second TC we use TC7 where CC0 is routed to WO0 (motor 3) and CC1 is routed to W01 (motor 4). The two TC are set to Normal Pulse-Width Modulation Operation (NPWM). This means the WO will be set at start to TC_s and cleared if the count of timer reaches the TC_{CC} value. We use the TC in oneshot mode this means the timer stops after overflow. The start value of the TC and the CC is calculated as followed:

  We define as clock source for the TC *GCLK_GENERATOR_3*. This is an $8MHz$ clock which means the timer gets incremented every $125ns$. From 2.4.2 we know that $t_p = 300\mu s$. So the start value t_s of the TC is:

$$TC_s = TC_{max} - \frac{300 * 10^{-6}}{\frac{1}{8 * 10^6}} = 2^{16} - \frac{300}{0,125} = 63136\ . \tag{A.1}$$

  The value of the CC now depends on the M_{out} of the motor mixer. The range of one motor m_x is from 1000 to 2000. On our $8MHz$ clock this will generate a time window

$$t_{on} = [t_{onMin}, t_{onMax}] \tag{A.2}$$

$$t_{onMin} = 1000 * 125ns = 1000 * 0,125 = 125\mu s \tag{A.3}$$

$$t_{onMax} = 2000 * 125ns = 2000 * 0,125 = 250\mu s \tag{A.4}$$

which is exactly what we need. The CC value is therefore:

$$TC_{CC} = TC_s + m_x \ . \tag{A.5}$$

- **USART**
  We configure SERCOM3 with 8N1 at a baudrate of 500000. As clock source we use the *GCLK_GENERATOR_2*. For sending and receiving we use interrupts. If data is received an interrupt is triggered and we copy the received data to the rx ring buffer. The write interrupt is triggered if the USART finished transmitting the data. If this happens and there is still data to transmit we proceed with transmitting data from tx buffer by creating a new write job for the USART.

  The driver also supports the transmission lock defined in 5.4.3. This is done by writing the data to the ring buffer when **beginWrite** was called. If **endWrite** is then called we trigger the same callback which is triggered by the write interrupt. This is useful to first enqueue some data without getting interrupted in normal program flow.

- **I/O Multiplexing**
  On *samd20j18a* the pins are by default handled as General Purpose Input Output (GIPO) pins. With the port multiplexer it is then possible to assign the pin to one defined peripheral function. See table6-1 in [Mic19] for the assignments for each pin. This restriction gets aggravated by **BMF055** because not all pins are available (routed to an actual pin) as well as some pins already get predefined peripheral function. Figure 3.1a shows which pins are available to the user. In Table A.1 the used mixer configuration is shown.

Table A.1: Used multiplexer configuration of *samd20j18a*.

pin	**peripheral function**	**description**
PA16	SERCOM1	SPI MOSI
PA17	SERCOM1	SPI SCKL
PA19	SERCOM1	SPI MISO
PA18	GIPO	SPI chip select accelerometer and magnetometer
PB03	GIPO	accelerometer INT1
PB23	GIPO	accelerometer INT2
PA27	GIPO	SPI chip select gyroscope
PA20	SERCOM3	tx pin
PA201	SERCOM3	rx pin
PA22	TC4	W0 motor1
PA23	TC4	W1 motor2
PB00	TC7	W0 motor3
PB01	TC7	W0 motor4

A.3.2 Driver

- **Accelerometer driver**
  The driver uses the same API as the *Data Stream example* from [Bos]. We use a range of $\pm 8g$ this means we get about $0.977mg/LSB$. The filter bandwidth is set to $125Hz$. The sensor raw value will be normalized for the IMU.

  The scale factor is
  $$s = \frac{1}{2^{13}/8} = \frac{1}{1024} \tag{A.6}$$

- **Gyroscope driver**
  The driver uses the same API as the *Data Stream example* from [Bos]. The range is set to $\pm 2000°/s$, the bandwidth of the internal filter is set to $116Hz$. The sensor raw value will be normalized for the IMU.

  The scale factor is
  $$s = \frac{1}{2^{15}/2000} = \frac{1}{16.4} \tag{A.7}$$

- **Configuration driver**
  To store the configuration, of the flight controller, we use the EEPROM module. A list of definded configuration parameters is shown in Table A.2.

Table A.2: List of firmware configuration parameter.

name	**description**
PILOTNAME	name displayed in *Betaflight Configurator*
MINTHROTTLE	minimal motor command to ESC
MAXTHROTTLE	maximal motor command to ESC
MINCOMMAND	minimal motor command to ESC if disarmed or throttle command < MINCHECK
MIDRC	receiver value for stick in neutral position (roll, pitch, yaw)
MINCHECK	minimal throttle command for motor mixer, if the throttle command is less then mincheck the motor mixer will output **min command**
MAXCHECK	maximal throttle command
YAW_DIRECTION	-1 or $+1$ depending on the motor arrangement
ACC_TRIM	trim values for the accelerometer
ARM_TIMEOUT_US	timeout between two rx packages
MAX_ARMING_ANGLE	maximal angle the quadcopter is able to be armed
rate_controller_config	rate controller configuration
RC_RATES	rc rates (factor the rate_command is multiplied with)
levelGain	gain for level mode
deciLevelAngleLimit	maximal angle for level mode
motorOneShot	determines if the ESCs are using oneShot protocol
blackboxEnabled	is blackbox enabled

List of Figures

List of Tables

Acronyms

AHRS Attitude and Heading Reference System. 8

ASF Atmel Software Framework. 90, 91

BLE Bluetooth Low Energy. 12, 25, 35, 42, 43, 61, 84

CC compare channels. 92, 93

CRC cyclic redundancy check. 13

DFLL Digital frequency locked loop. 91

DSP Digital signal processing. 26

EEPROM Electrically Erasable Programmable Read-only Memory. 91, 94

ESC Electronic Speed Controller. 12, 13, 29, 30, 34, 39, 41, 44, 46, 55, 75, 76, 94, 96

FFT Fast Fourier transform. 58

FPU Floating Point Unit. 26

GIPO General Purpose Input Output. 93

HAL Hardware Abstraction Layer. 44

I2C Inter-Integrated Circuit. 32, 43, 89

IMU Inertial Measurement Unit. 7, 8, 14, 16, 44, 46, 67, 80, 81, 83–85, 87, 88, 94, 96, 98

IoT Internet of Things. 1, 12, 26

LED light-emitting diode. 37

LiPo Lithium Polymer. 31

LTMv2 LightTelemetry v2. 43

MAV Micro Aerial Vehicles. 67

MCU microcontroller unit. 1, 2, 12, 15, 16, 18, 19, 21, 24, 42, 44, 90

MSP Multiwii Serial Protocol. 13, 14, 16, 49–51, 57, 61, 62, 65, 67, 88, 95, 96, 98

NFC Near Field Communication. 26

NPWM Normal Pulse-Width Modulation Operation. 92

NVM Non-volatile memory. 91, 92, 97

PC Personal Computer. 1

PDB Power distribution board. 30, 34

PID Proportional-Integral-Derivative. 8, 9, 15, 16, 41, 46, 58, 67, 77, 80, 88, 96

PPM Pulse Position Modulation. 15

PS Play Station. 42

PWM Pulse Width Modulation. 13, 15, 75, 91

RC Radio Controlled. 13

ROS Robot Operating System. 35, 42, 67, 75, 76, 82, 88, 96

Rx Receiver. 15

SiP System in Package. 2, 21, 24, 42

SITL Software In the Loop. 2, 16, 24, 65–67, 75–77, 87, 88, 96

SPI Serial Peripheral Interface. 15, 21, 22, 91

SWD Serial Wire Debug. 90

TC Timer/Counter. 92

TCP Transmission Control Protocol. 16, 49, 61, 65, 75

TOF Time of Flight. 20, 26, 32, 43

TPA Throttle PID Attenuation. 15

Tx Transmitter. 15

UART Universal Asynchronous Receiver Transmitter. 15, 43

UDP User Datagram Protocol. 75

USART Universal Synchronous and Asynchronous Serial Receiver and Transmitter. 49, 93

USB Universal Serial Bus. 39

WO waveform output. 92

Bibliography

[Arda] ArduPilot. ArduPilot Hardware List. URL: https://ardupilot.org/copter/docs/common-autopilots.html#common-autopilots. [Online; last accessed: 07-11-2020].

[Ardb] ArduPilot. Open Source Drone Software. Versatile, Trusted, Open. ArduPilot. URL: https://ardupilot.org/. [Online; last accessed: 07-11-2020].

[Arm] Armstart. IBDAP - User Manual. URL: https://cdn-shop.adafruit.com/product-files/2764/c4119_IBDAP-User-Manual.pdf. [Online; last accessed: 02-04-2020].

[Baca] Backyard Robotics. On ESC Protocols Part I. URL: https://backyardrobotics.eu/2018/03/02/on-esc-protocols/. [Online; last accessed: 14-11-2020].

[Bacb] Backyard Robotics. On ESC Protocols Part II. URL: https://backyardrobotics.eu/2018/03/14/on-esc-protocols-part-ii/. [Online; last accessed: 14-11-2020].

[Bas] Baseflight. Baseflight. URL: https://github.com/multiwii/baseflight/. [Online; last accessed: 26-05-2020].

[Beta] Betaflight. BetaFlight. URL: https://github.com/betaflight/betaflight/wiki. [Online; last accessed: 06-09-2018].

[Betb] Betaflight. Betaflight Blackbox Explorer. URL: https://github.com/betaflight/blackbox-log-viewer. [Online; last accessed: 07-11-2020].

[Betc] Betaflight. Betaflight Configurator. URL: https://github.com/betaflight/betaflight-configurator. [Online; last accessed: 07-11-2020].

[Betd] Betaflight. Betaflight Msp Code. URL: https://github.com/betaflight/betaflight/tree/master/src/main/msp. [Online; last accessed: 16-04-2021].

[Bete] Betaflight. Blackbox logging internals. URL: https://cleanflight.readthedocs.io/en/stable/development/Blackbox%20Internals/. [Online; last accessed: 07-11-2020].

[Blo] Lukas Blocher. BMF055-flight-controller. URL: https://github.com/NightHawk32/BMF055-flight-controller. [Online; last accessed: 06-09-2018].

[Bos] BoschSensortec. Example Project – Data Stream. URL: https://www.bosch-sensortec.com/media/boschsensortec/downloads/application_notes_1/bst-bmf055-ex001-01.pdf. [Online; last accessed: 2-04-2020].

[Bos15] BoschSensortec. BST-BMF055-DS000-01. URL: https://www.bosch-sensortec.com/media/boschsensortec/downloads/datasheets/bst-bmf055-ds000.pdf, 11 2015. [Online; last accessed: 02-04-2020].

[Bos16] BoschSensortec. BST-BMF055-AN001-01. URL: https://www.bosch-sensortec.com/media/boschsensortec/downloads/application_notes_1/bst-bmf055-an001.pdf, 01 2016. [Online; last accessed: 02-04-2020].

[Cha] Charles Tytler. Modeling Vehicle Dynamics – Quadcopter Equations of Motion. URL: https://charlestytler.com/quadcopter-equations-motion/. [Online; last accessed: 30-10-2020].

[CHR] CHRobotics LLC. Understanding Euler Angles. URL: http://www.chrobotics.com/library/understanding-euler-angles. [Online; last accessed: 27-10-2020].

[Cle] Cleanflight Team. Cleanflight Code. URL: https://github.com/cleanflight/cleanflight. [Online; last accessed: 07-11-2020].

[DAL] DALPROP. DAL 5045 v2 Bullnose Propeller (2x Cw 2x CCW) Grün. URL: https://www.fpv24.com/de/dalrc/dal-5045-v2-bullnose-propeller-gruen-2xcw-2xccw. [Online; last accessed: 23-05-2021].

[DKMV16] Jan Dentler, Somasundar Kannan, Miguel Angel Olivares Mendez, and Holger Voos. A real-time model predictive position control with collision avoidance for commercial low-cost quadrotors. In *2016 IEEE Conference on Control Applications (CCA)*, pages 519–525, 2016.

[Dra18] Rainer Drath. Die Idee hinter Industrie 4.0: Mehr als Technik. In *iX Special 2018 - Industrial Internet of Things*, pages 8–11. Heise Medien, 2018. ISBN-9783957882066.

[EST+18] Emad Samuel Malki Ebeid, Martin Skriver, Kristian Terkildsen, Kjeld Jensen, and Ulrik Schultz. A survey of open-source uav flight controllers and flight simulators. *Microprocessors and Microsystems*, 61, 05 2018.

[FBAS16] Fadri Furrer, Michael Burri, Markus Achtelik, and Roland Siegwart. *RotorS – A Modular Gazebo MAV Simulator Framework*, volume 625, pages 595–625. 01 2016.

[För15] Julian Förster. System identification of the crazyflie 2.0 nano quadrocopter. B.S. thesis, ETH Zurich, 2015.

[GMM12] Nils Gageik, Thilo Müller, and Sergio Montenegro. Obstacle detection and collision avoidance using ultrasonic distance sensors for an autonomous quadrocopter. *University of Wurzburg, Aerospace information Technologhy (germany) Wurzburg*, pages 3–23, 2012.

[Hoba] Hobbywing. Hobbywing X-Rotor 30A Micro 2-4S ESC with BLHeli-S Dshot600 (Opto). URL: `https://hobbyking.com/de_de/hobbywing-xrotor-30a-micro-2-4s-blheli-s-dshot600.html`. [Online; last accessed: 23-05-2021].

[Hobb] Hobbywing. XRotor 2205. URL: `https://www.hobbywing.com/goods.php?id=522&filter_attr=7591.0.0.0.0`. Online; last accessed: 23-05-2021].

[Hor] Horizon Hobby LLC. Specification for Spektrum Remote Receiver Interfacing. URL: `https://www.spektrumrc.com/ProdInfo/Files/Remote%20Receiver%20Interfacing%20Rev%20A.pdf`. [Online; last accessed: 30-03-2020].

[Ign90] Mario B Ignagni. Optimal strapdown attitude integration algorithms. *Journal of Guidance, Control, and Dynamics*, 13(2):363–369, 1990.

[Ina] Inav. Multiwii Serial Protocol Version 2. URL: `https://github.com/iNavFlight/inav/wiki/MSP-V2`. [Online; last accessed: 16-10-2020].

[Joe10] Decuir Joe. Bluetooth 4.0: low energy. *Cambridge, UK: Cambridge Silicon Radio SR plc*, 16, 2010.

[Kui99] Jack B Kuipers. *Quaternions and rotation sequences: a primer with applications to orbits, aerospace, and virtual reality*. Princeton university press, 1999.

[Lia] Oscar Liang. F1, F3, F4, F7 and H7 Flight Controller Explained. URL: `https://oscarliang.com/f1-f3-f4-flight-controller/`. [Online; last accessed: 07-11-2020].

[Lig] LightingHobby. LHI H280 FPV Race Quadcopter Rahmen von Full Carbon Faser Rahmen. URL: https://www.amazon.com/-/de/dp/B010FIIQL6/ref=cm_cr_arp_d_product_top. [Online; last accessed: 23-05-2021].

[LN16] Carlos Luis and Jérôme Le Ny. Design of a trajectory tracking controller for a nanoquadcopter. *arXiv preprint arXiv:1608.05786*, 2016.

[Luu11] Teppo Luukkonen. Modelling and control of quadcopter. *Independent research project in applied mathematics, Espoo*, 22, 2011.

[Mad] Sebastian O.H. Madgwick. An efficient orientation filter for inertial andinertial/magnetic sensor arrays. URL: https://www.x-io.co.uk/res/doc/madgwick_internal_report.pdf. [Online; last accessed: 6-09-2018].

[MDCDW+17] Kimberly McGuire, Guido De Croon, Christophe De Wagter, Karl Tuyls, and Hilbert Kappen. Efficient optical flow and stereo vision for velocity estimation and obstacle avoidance on an autonomous pocket drone. *IEEE Robotics and Automation Letters*, 2(2):1070–1076, 2017.

[Mel12] Daniel Warren Mellinger. Trajectory generation and control for quadrotors. Dissertation, University of Pennsylvania, 2012.

[Mic19] Microchip Technology Inc., 2355 West Chandler Blvd. Chandler, Arizona, USA 85224-6199. *Low-Power, 32-bit Cortex-M0+ MCUs with 12-bit ADC, 10-bit DAC, 256-Channel PTC, RTC, and SERCOM*, revision c - 11/2019 edition, 11 2019. URL: http://ww1.microchip.com/downloads/en/DeviceDoc/SAM_D20_%20Family_Datasheet_DS60001504C.pdf [Online; last accessed: 24-07-2021].

[Mul] MultiWii. MultiWii. URL: http://www.multiwii.com/. [Online; last accessed: 26-05-2020].

[Ope] Open Source Robotics Foundation. Gazebo. URL: http://gazebosim.org/. [Online; last accessed: 16-04-2021].

[Par] Particle Industries, Inc. Xenon Datasheet. URL: https://docs.particle.io/datasheets/discontinued/xenon-datasheet/. [Online; last accessed: 16-04-2021].

[Pol] Pololu Corporation. VL53L0X library for Arduino. URL: https://github.com/pololu/vl53l0x-arduino. [Online; last accessed: 23-05-2021].

[PP16] Viswanadhapalli Praveen and Anju Pillai. Modeling and simulation of quadcopter using pid controller. *International Journal of Control Theory and Applications*, 9(15):7151–7158, 2016.

[PX4a] PX4 Team. Pixhawk 4. URL: `https://docs.px4.io/master/en/flight_controller/pixhawk4.html`. [Online; last accessed: 07-11-2020].

[PX4b] PX4 Team. Pixhawk 4 - ROS (Robot Operating System). URL: `https://docs.px4.io/master/en/ros/`. [Online; last accessed: 16-04-2021].

[QGS15] Morgan Quigley, Brian Gerkey, and William D Smart. *Programming Robots with ROS: a practical introduction to the Robot Operating System*. O'Reilly Media, Inc., 2015.

[Rat] Dominic Rath. Open On-Chip Debugger. URL: `http://openocd.org/`. [Online; last accessed: 30-03-2020].

[SAI18] G. Silano, E. Aucone, and L. Iannelli. Crazys: A software-in-the-loop platform for the crazyflie 2.0 nano-quadcopter. In *2018 26th Mediterranean Conference on Control and Automation (MED)*, pages 1–6, 2018.

[Sch18] Uwe Schulze. Low Power Wild Area Networks fürs IIoT. In *iX Special 2018 - Industrial Internet of Things*, pages 96–101. Heise Medien, 2018. ISBN-9783957882066.

[Spa] Sparkfun. RFduino - Simblee DIP. URL: `https://www.sparkfun.com/products/retired/13768`. [Online; last accessed: 01-08-2021].

[ssk] sskaug. BLHeli manual SiLabs Rev14.x. URL: `https://github.com/bitdump/BLHeli/blob/master/SiLabs/BLHeli%20manual%20SiLabs%20Rev14.x.pdf`. [Online; last accessed: 30-03-2020].

[STM16] STMicroelectronics. *World smallest Time-of-Flight ranging and gesture detection sensor Application Programming Interface*, docid029105 rev 1 edition, 5 2016. URL: `https://www.st.com/resource/en/user_manual/dm00279088-world-smallest-timeofflight-ranging-and-gesture-detection-sensor-application-programming-interface-stmicroelectronics.pdf` [Online; last accessed: 02-04-2020].

[STM18] STMicroelectronics. *World's smallest Time-of-Flight ranging and gesture detection sensor*, docid029104 rev 2 edition, 4 2018. URL: `https://www.st.com/resource/en/datasheet/vl53l0x.pdf` [Online; last accessed: 02-04-2020].

[Sus] Sushmita Venugopalan. How to stably spin a cuboid. URL: https://www.imsc.res.in/~knr/past/facets16/tumbling.pdf. [Online; last accessed: 30-10-2020].

[Tema] Alexander Temper. bmflight. URL: https://github.com/AlexanderTemper/bmflight. [Online; last accessed: 27-05-2021].

[Temb] Alexander Temper. DroneSimulation. URL: https://github.com/AlexanderTemper/DroneSimulation. Online; last accessed: 27-05-2021].

[Temc] Alexander Temper. Simulation Video. URL: https://drive.google.com/file/d/1ig9HJmgjAXggvWgCfPK_qzO1nL-IxSxJ/view. [Online; last accessed: 09-06-2021].

[Temd] Alexander Temper. spinningDrone. URL: https://github.com/AlexanderTemper/spinningDrone/. [Online; last accessed: 27-05-2021].

[Teme] Alexander Temper. spinningDroneGateway. URL: https://github.com/AlexanderTemper/spinningDroneGateway. [Online; last accessed: 27-05-2021].

[The] The Linux Foundation. Zephyr Project. URL: https://zephyrproject.org/. [Online; last accessed: 16-04-2021].

[Tho20] Puchinger Thomas. Iot sensor swarm for agricultural micro climate measurement. M.S. thesis, TU Wien, 2020.

[Tri] Trimble Inc. SketchUp. URL: https://www.sketchup.com/. [Online; last accessed: 16-04-2021].

[ZZTL19] Lanxiang Zheng, Ping Zhang, Jia Tan, and Fang Li. The obstacle detection method of uav based on 2d lidar. *IEEE Access*, 7:163437–163448, 2019.