Quadrocopter Flight-Control Design using SCCharts

Lewe Andersen

Bachelor-Thesis
2015

Prof. Dr. Reinhard von Hanxleden
Real-Time and Embedded Systems
Department of Computer Science
Kiel University

Advised by
Dipl.-Inf. Steven Smyth and Dipl.-Inf. Christian Motika

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Kiel,

Abstract

This thesis presents the development of a flight-control system for a quadrocopter that was built during a student project. The system was implemented both in C/C++ and the visual synchronous language SCCharts. Both implementations will be described and compared to each other to evaluate the usability of SCCharts for this kind of project.

Contents

List of Figures

List of Tables

Abbreviations and Symbol Index

UAV	Unmanned Aerial Vehicle
ESC	Electronic Speed Controller
DMP	Digital Motion Processor
KIELER	Kiel Integrated Environment for Layout Eclipse RichClient
IDE	Integrated Development Environment
I^{2}C	Inter Integrated Circuit
i2cDevLib	I^{2}C Device Library

Chapter 1

Introduction

Every step we take in today's world is followed by various computer system. May it be the security camera of a building we pass by or the smartphone we carry that updates its location information. Even if we wait for a lift we are most of the time not conscious of the computer that is working to fulfil our wishes. Those systems, called embedded systems, take control over more and more parts of our daily life. As example one could look on cars. Every new model has even more new computer controlled features as described in an article by Klaus Bengler et al. [BDF+14]. This gives a clear perspective on how important the use of embedded systems in today's vehicles has become and even gives a small insight on how much more importance it will get.

Unmanned Aerial Vehicles (UAVs) such as military or parcel delivery drones also follow this trend. Especially the quadrocopter, a vehicle similar to a miniature helicopter driven by four rotors, gains popularity in economic and private usage.

Those quadrocopters depend on various different embedded systems. One of those systems, the so called *flight-control*, is responsible to ensure a stable flight. Without this system the quadrocopter would not even be able to fly and the fact that there already are flying quadrocopters shows that working flight-control systems exist and are fully functional.

However after the introduction of the visual synchronous language *Sequentially Constructive Charts* (SCCharts) by Hanxleden et al. [HDM+13] it is interesting to see whether it is possible to implement such a flight-control with SCCharts and, if possible, to determine the degree of complexity of the resulting program.

To evaluate these questions a group project was done. The main goals of this project were to create a working flight-control for a quadrocopter and to develop an obstacle avoidance system for it. Another target was to create a simulation of a quadrocopter-environment making it possible to test the systems virtually before they are tested in the reality.

This thesis will focus on the development of the flight-control in this project and implementations with C-Code and SCCharts. After this a comparison of both implementations will be given.

The following sections permit an insight on what the flight-control is needed for on the one side, on the other side providing an overview of the quadrocopter used with this project and in the end gives a short introduction to SCCharts. An overview about further chapters will be given in Section 1.4.

The other parts of the project can be further reviewed in the work on obstacle avoidance [Mac15] and the work on simulations [Pei15].

1.1 Problem Description

Unlike a plane, a quadrocopter does not fly by airstreams on its wings, but by the power of its four rotors. This is why a quadrocopter has no ability for a stable flight by itself. It is highly influenced by air movements, such as wind or some kind of turmoil, or unbalanced weight. The flight-control of a quadrocopter has the main task to detect environmental influences and to calculate how the throttle on every rotor has to be changed to negate them. For this task a sensor is needed to detect the angles on the three axes of the quadrocopter relative to the earth level. With this sensor the flight-control can detect the quadrocopters deviation on each axis to the desired angle. Based on this deviation the flight-control calculates the amount of throttle-change that has to be forwarded to each rotor to get back to a levelled position. Further it is possible to achieve a stable movement of the copter by changing the desired angles. This makes the flight-control a central system of a quadrocopter.

1.2 Quadrocopter

At the beginning of the project the first decision was, what kind of quadrocopter were the best to use: it was either possible to buy a finished set or to build one out of separate parts. It was decided not to buy a finished set because it is always combined with a functional flight-control, which is not guaranteed to be rewritable, or if the microcontroller might be compatible with SCChart-programs. Thus the decision was made to build a quadrocopter out of separate parts. The selection of all these parts was made with regard to provide all needed functions with as few parts as possible with the lowest possible price.

So the quadrocopters frame is the *F330 Glass Fiber Mini Quadcopter Frame*[1] and it is driven by four *Turnigy Aerodrive SK3 2822-1275 Brushless Outrunner motors*. Each of these motors is connected to an Electronic Speed Controller (ESC) to control their throttle. Four *Turnigy Plush 12amp (2A BEC) BESC* are used for this task. The next item is the position sensor *MPU9150* by IvenSense, which provides the gyroscope, accelerometer and a magnetometer that are needed to calculate the angular position of the quadrocopter. This sensor also comes with an onboard Digital Motion Processor (DMP), that is capable of calculating the needed angles out of the sensor raw values by itself. This removes some of

[1]During the building process it was seen that the chosen frame does not provide enough possibilities to connect all the parts to it, so some connection parts were created by a 3D-printer.

the calculation process from the main control unit. The resulting benefit of this is saving time, as described in Chapter 4.

Next to this it is needed to get the possibility to communicate with the quadrocopter in flight. For this task the *Aukru HC-06* Bluetooth module is used. It provides the quadrocopter with a serial communication port that on one side will be used to send commands to the copter and on the other side to receive data from the copter, such as the current position.

The sensors used to add the obstacle avoidance are 10 Ultrasonic-Modules *HC-SR04*. They work by emitting an ultrasonic-signal and wait for its response. The waited time is then used to calculate the distance. However to check a distance of 2 meters on this way consumes around 12 microseconds. This leads into 120 microseconds of latency if 10 sensors are used. To reduce this latency the task to determine the ultrasonic sensor values is taken over by two separate controllers. One is an *Arduino Nano*, the other one is an *Arduino Mini* board, which both communicate the data via a serial port to the main controller.

The main controller used for this project is an *Arduino Mega 2560 Board*. This board was chosen, because it provides 54 digital I/O pins and 16 analog inputs, including 4 separated ports for serial communication. This multitude is needed to connect all sensors to it and the serial ports are needed, because it is necessary to communicate with three different modules as described earlier in this section.

The system is powered by the *Turnigy 2200mAh 3S 20C Lipo Pack*. This battery delivers a voltage of 12V and is directly connected to the ESCs that include a voltage divider and give themselves an output of 5V, which is used to power the Arduino.

1.3 Sequentially Constructive Charts

Sequentially Constructive Charts (SCCharts) is a visual synchronous language designed to specify safety-critical reactive systems such as the flight-control of a quadrocopter. SCCharts uses the recently presented synchronous model of computation [HDM+14] and a statechart notation derived from SyncCharts [And03]. An implementation is given in the Kiel Integrated Environment for Layout Eclipse RichClient (KIELER) project with a compiler that is among others possible to translate the designed Charts into C-Code. This C-Code consists of two methods:

reset()
This method is used to reset the chart. It needs to be invoked at the begin of the process and makes sure that the model is started at the initial state.

tick()
This function is meant to be called repeatedly as long as the modelled system is running. The inputs declared in the model are parameters for the function and based on the

current state of it the following state is determined and the declared outputs are set according to this state.

These two functions are very similar to the two functions that are needed for an Arduino sketch as described in Chapter 3. Thus all that is to do, to run a SCChart model on an Arduino, is to invoke the reset-function in the setup-function of the Arduino and then forward the determined sensor values as inputs for the tick-function inside the loop.

1.4 Outline

Chapter 2 will discuss related work. It will show how other projects have worked with the topic and explain why this work does not follow those approaches.

Chapter 3 describes the used technology during the working process to provide an insight on how results were achieved and to give a better opportunity to do some deeper research on this topic.

Chapter 4 will describe the process of developing the flight-control, starting with theory about quadrocopters. It will be explained how it is possible for a quadrocopter to fly without spinning around, like a helicopter would do without its tail-rotor, and how it is navigated. After that the usage of the Position sensor will be described. It will be explained, how the needed data is collected and what problems occurred with it during the project. Thereafter the used controller will be described. It will be started with the theory behind it and ended with the methods used to adjust it onto the quadrocopter used. The following part of this chapter will be used to describe the developed protocol for the manual flight abilities of the quadrocopter. What signals are transmitted and how the system reacts to them. The last Section will give an insight on the biggest problems that were faced during the development process and what was tried to solve them.

Chapter 5 will show how the designed flight-control was implemented. First the approach in C/C++ code will be shown and after that the implementation in SCCharts will be explained.

Chapter 6 will evaluate the work, focused on a comparison of the SCChart approach with the C/C++ implementation.

Chapter 7 will then be a conclusion including a summery of the work and its results and an estimate on future work in this topic.

Chapter 2

Related Work

As mentioned in Chapter 1 the quadrocopter is an UAV with a growing public interest and also the flight-control is a system that was already designed by others:

A. Zul Azfar et al. [AH11] designed a flight-control system for a quadrocopter by using a PID-Controller, to determine the throttle-change that is needed for a stable flight. Azfar et al. used a new method to calculated the sensor data used by the PID-Controller, which is the main topic of their work. The PID-Controller will also be used for this thesis and will be described in detail in Section 4.3, but the method to calculate the sensor data will not be considered as explained in Section 4.2.

Further S. Bouabdallah et al. [BNS04] developed a flight-control for an indoor micro quadrotor. In their article they discuss the difference of the PID-Controller to another controller that is following a linear-quadratic regulation approach. As mentioned earlier this thesis uses the PID-Controller. Due to a lack of time it was not possible to consider both approaches.

Z. T. Dydek et al. [DAL12] designed a flight-control that is beside providing a stable flight also capable of reacting on system errors such as component failure or physical damage. This is a needed feature for a safety-critical system such as a quadrocopter. However a reaction on system errors was not part of this thesis.

Last to mention is the flight-control designed by S. Klose et al. [KWA+10]. Their system works without an onboard position sensor, but with external stereo camera measurements. The camera detects the current position of the quadrocopter and the flight-control calculates the needed throttle-change based on this data. However our system was meant to work without external calculations, thus it was necessary to use an onboard sensor.

While the related work mainly concentrates on special features of the flight-control our work uses the most simple way to design this needed system. Our goal was not to create a new kind of flight-control, but to demonstrate the usability of SCCharts to implement these system.

Chapter 3

Used Technologies

3.1 Arduino

Arduino is an open source hardware and software development platform since 2005. It includes several different microcontroller boards and compatible extensions to them. One board consists of a processor and a multitude of analogue and digital input and output pins. To program this board a free to download Arduino Integrated Development Environment (IDE) exists that comes with the necessary libraries to use all features of the Arduino board. The IDE includes a compiler and tools to upload the developed program onto the Arduino board. The programming language used to write an Arduino program, that is called a *sketch*, is based on C/C++. It has small extensions to make the use of hardware specific features possible. The most simple Arduino project consists of one *.ino*-file. Both the project-folder and the .ino-file need to have the same name. This determines, which file is the main-file of the project. This main-file has to contain at least a setup-method, which is called at the start-up of the board, and a loop-method, which will be constantly called after the setup-method has finished as long as the Arduino-board is powered. This project can be extended by other .ino-files that do not need to be included in the main-file. It is also possible to use C or C++ files, but using them requires an inclusion in the .ino-files.

3.2 I^{2}C Device Library

The MPU9150 that was used in our project communicates via the Inter Integrated Circuit (I^{2}C) with the Arduino board. For devices that use this communication the open source I^{2}C Device Library (i2cDevLib) was developed. It contains a basic library for I^{2}C communication as well as a multitude of libraries for different devices all separated for the use with different microcontrollers. It is used to simplify the communication process. Without the library the process to request data from the position sensor consists of sending a certain byte-sequence to certain request-pins of the sensor and thereafter reading the data from certain output-pins. The library provides the user clear named functions to execute the desired requests. Additionally the i2cDevLib contains an example for using the DMP of the MPU6050, which is the predecessor of the MPU9150. The DMP can be used to calculate the needed sensor values out of its raw data. The advantages of this are explained in Section 4.2. It is required to use an example for the older sensor, because there is no equivalent for the MPU9150. However this is no disadvantage - the example works without changes for the MPU9150.

3.3 Autodesk Tinkercad

Tinkercad is a free to use online application. It provides a drag and drop based editor to create 3D-models that can be exported for 3D-printing. This software was used to design connecting pieces that were needed to attach all parts to the chosen frame. For example the chosen microcontroller did not fit the screw holes of the frame. A 3D-model of a combination part was designed with tinkercad and 3D-printed to fix this problem.

Chapter 4

Flight-Control Design

To design a flight-control system it is necessary to understand which underlying systems represent a flight-control and how they work. Additionally knowledge about the piloting of a quadrocopter is required to use the values calculated by the flight-control the right way. The following chapter initially explains the structure of a quadrocopter and how it can be piloted. Afterwards the components of the flight-control will be described. Ultimately an insight on the problems faced during the development will be given and how we tried to solve them.

4.1 Quadrocopter Principles

The typical quadrocopter consists of a cross-like frame with one motor attached to each end of it. The control-electronics such as the position sensor and the microcontroller are placed in the centre of this cross. The thrust that is created by the rotors, which are connected to the four motors, provides the quadrocopter with the ability to fly. Additionally a spinning rotor induces a torque onto the quadrocopter. Without further intervention this force would let the quadrocopter spin uncontrollably. Thus to eliminate this torque the fact that a quadrocopter has more than one rotor is used. So two of these rotors turn clockwise and the other two turn counter clockwise. On this way two torques are induces that eliminate each other. The next thing to mention is that the only controllable values on a basic quadrocopter are the throttles of the different motors. By changing these values the quadrocopter is balanced and piloted. Thus the task for a flight-control is to calculated how the throttle of each motor has to be changed to achieve the desired behaviour. To do these calculations two things have to be known about the underlying quadrocopter: Its orientation and how the throttles have to be changed to acquire a certain movement.

4.1.1 Quadrocopter Orientation

The orientation of a quadrocopter decides what part of it is the front regarding to its flight direction and with this how the throttle of the motors has to be changed to perform the tilting movements. There are two different possibilities:

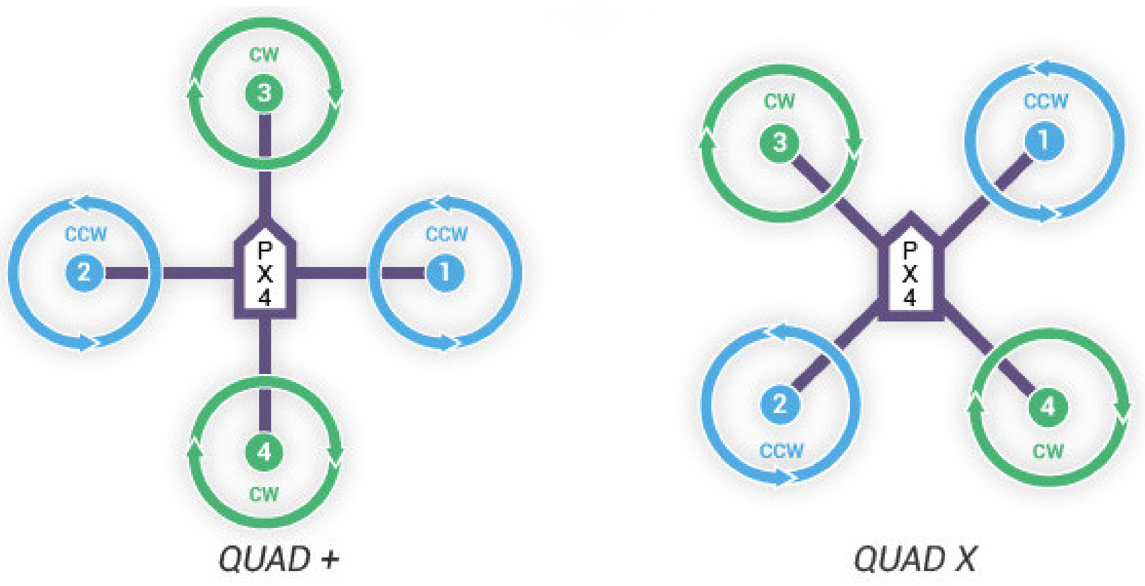

Figure 4.1. Quadrocopter orientations (soure: www.axnzero.com)

Quadrocopter + configuration
The + configuration as shown on the left side of Figure 4.1 has on motor leading the flight direction of the quadrocopter. It is called + configuration, because the frame of the quadrocopter forms a +. The advantage of this configuration is that the tilting movements can be performed only by changing the throttle of two motors instead of all four. The disadvantage of this configuration is that the view of a camera facing the flight direction would always be restricted by on motor.

Quadrocopter X configuration
The X configuration as shown on the right side of Figure 4.1 has the frame axis arranged in a 45 degree angle to the flight axis. On this way the frame forms a X. The advantage of this configuration is the open view a camera would have that is aligned to the flight direction. The disadvantage is that for every possible movement the throttle of all four motors needs to be adjusted.

In this project we decided to use the X-configuration, because we wanted to make it easy to extend the quadrocopter with a camera.

4.1.2 Quadrocopter Movement

As described in Section 4.1.1 the chosen orientation determines how the throttle of each motor has to be changed to perform a certain movement. The possible movements are

Table 4.1. How to perform a certain movement with a quadrocopter

Movement	Throttle-Change
Gain Height	1+ 2+ 3+ 4+
Loose Height	1- 2- 3- 4-
Tilt Forward	1- 2+ 3- 4+
Tilt Backward	1+ 2- 3+ 4-
Tilt Left	1+ 2- 3- 4+
Tilt Right	1- 2+ 3+ 4-
Turn Clockwise	1+ 2+ 3- 4-
Turn Counter Clockwise	1- 2- 3+ 4+

gaining and losing height together with the rotations around the three axes of the quadrocopter. As shown in Figure 4.2 a rotation around the x-axis (the axis that is facing the flight-direction of the quadrocopter) is called a roll, a rotation around the y-axis is called pitch and a rotation around the z-axis is called yaw. Before a movement can be done an amount of throttle-change has to be calculated. This value determines how strong the resulting movement is. The process of calculating this throttle-change will be described in Section 4.3. The important thing is that the calculated amount is used on every motor. With the chosen X configuration a movement always requires to change the throttle of all four motors and it is necessary to change all of them by the same absolute value, otherwise two movements will be combined. As example if the quadrocopter is meant to gain height by a calculated throttle-change of 5, but the change of 5 is only given to the two front motors and the throttle of the two rear motors is only change by an amount of 4, this would not just lead in a gain of height but also in a tilting backwards.

Table 4.1 gives an overview about how the throttle of the motors has to be changed to perform a certain movement. The enumeration of the motors follows the example of the X configuration in Figure 4.1.

4.2 Working with Position Sensors

The main task of a flight-control is to provide a stable flight. To do this two things have to be done: Firstly it is needed to detect whether the quadrocopter does an unwanted movement. Secondly the throttle-change for each motor has to be calculated that is needed to stop the detected movement and get back to the desired position. The position sensor is used to detect such unwanted movements. It measures the angular position of the quadrocopter regarding to the earth level. With this information it is possible to calculate the difference between the current position and the desired one. This difference is called the error that needs to be corrected.

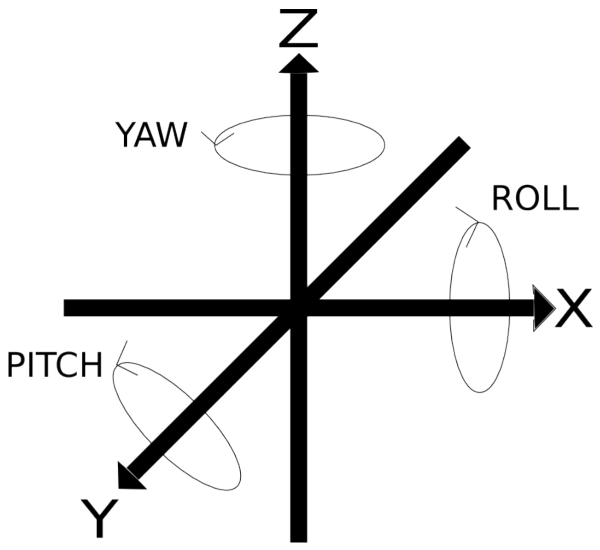

Figure 4.2. Yaw, pitch, and roll

4.2.1 Sensor Fusion

The common position sensor consists of two parts: a *accelerometer* that detects the current acceleration on every axis and a *gyroscope* that detects the angular velocity. As mentioned in Chapter 1 it is desired to determine the angular position of the quadrocopter by one angle for the current yaw, one for pitch and one for roll. The problem is that none of the two mentioned sensors can be used to get accurate readings of these angles. The accelerometer can be used to determine the direction to the ground be detecting the gravity, but as soon as the Quadrocopter is moving and adds other accelerations to the sensors readings the process of detecting the gravity becomes difficult. The data of the gyroscope on the other hand can be integrated over time to get the angular changes regarding to the starting position. The problem with this approach is that measurement errors would also integrate into the estimated angular change which would build up an increasing error, especially because most gyroscopes have a small bias. The most common solution for this problem is to fuse the readings of these two sensors using a kalman filter. With this technique the gyroscope data is used to determine quick angular changes and the accelerometer data is used to correct the increasing error. The problem of this approach is that the detection of gravity can not be used to correct a yaw-angle. For this process a magnetometer is needed including another filtering process. The fact that the used position sensor has an onboard DMP that includes algorithms to do this needed sensor fusion gives an easy possibility to avoid these calculation on the main microcontroller. Thus it was decided to use this DMP for the flight-control of this project.

4.2.2 Using the Digital Motion Processor

The DMP provides a big advantage, but the official data sheet of the MPU9150 does not offer any information about how to use the DMP. The data sheet just includes a small subsection about what this chip is capable of. An advanced research showed that the needed information was only available for registered developers of the producer. The only help that was found, was the in Section 3.2 mentioned open source library i2cDevLib. It provides an example for the DMP usage of the MPU6050, which is the predecessor of the MPU9150. The MPU6050 is the same sensor as the MPU9150 with the difference that it is missing a magnetometer. Thus the given example is without any changes usable for the MPU9150. Despite the fact that the example does not make use of a magnetometer the DMP is still capable of calculating an accurate yaw angle. To achieve this the DMP makes use of algorithms that were not published by the producer. Thus this process can not be explained here.

To use the DMP with the given library the first step is to determine the sensor specific offsets. Every sensor comes with different offsets that have to be determined to get the right angles. To simplify this task the developers of the i2cDevLib have created an Arduino sketch. To make this work the sensor has to be placed in a levelled position. The sketch reads the raw values of the accelerometer and the gyroscope and calculates their averages of 1000 readings to overcome measurement errors. A perfectly configured sensor would return zero values on all three gyroscope axes and on the x- and y-axis of the accelerometer. Only the accelerometer value of the z-axis would return the value that is representing the acceleration of gravity. The sketch calculates offset values based on the read averages and then repeats this process. As soon as the read averages reach a predefined range around the desired value the sketch returns the used offsets to achieve these readings. In our case the sensor was already attached to the quadrocopter in a faced down position. To make the sketch work the quadrocopter had to be in a levelled position and the algebraic sign of the desired value had to be changed from plus to minus. With the so calculated offsets the example for DMP use given by the i2cDevLib, could be used to determine exact yaw, pitch and roll angles.

4.2.3 The Yaw-Angle Problem

As mentioned in Section 4.2.1 the usual way to determine the yaw-angle includes a magnetometer into its calculation. The fact that the used DMP does not include the magnetometer readings results in a different zero point of the yaw angle for every system start-up. The fact that the flight-control tries to stabilize the quadrocopter at zero degrees would result in a turning movement at the launch until the zero angle is reached. To overcome this an offset of the yaw-angle is determined in the beginning and will be subtracted from the sensor data. Another behaviour of this data that needs explanation is

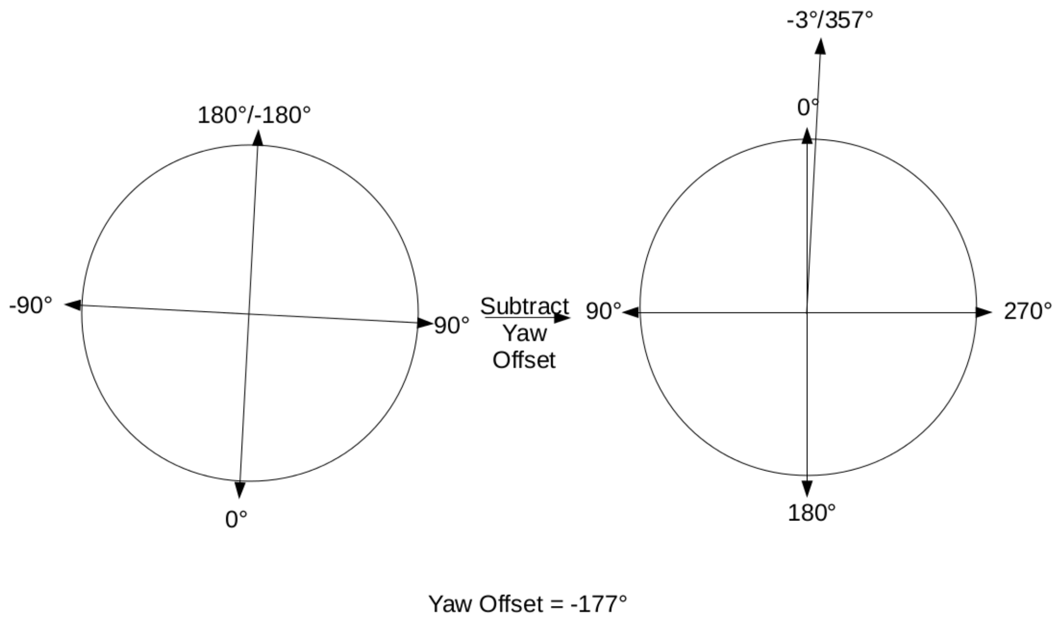

Figure 4.3. Correcting the yaw angle by an offset.

its scale. A clockwise yaw-rotation results in a negative yaw-angle and a counter clockwise rotation results into a positive yaw-angle. In both directions the measured data scales up to an absolute value of 180 degrees. So we have got +180 degrees and -180 degrees at the same point. This creates a gap of 360 degrees in the scale of the yaw-angle. Due to the explained offset this point will also be shifted to another place, as shown in Figure 4.3.

If the calculated offset at the beginning, for example, is 177 degrees, a counter clockwise turn of 5 degrees would not result in a measured value of $182 - 177 = 5$ but into $-178 - 177 = -355$. This means that the flight-control does not calculate with an error of 5 degrees but with an error of -355 degrees, which would result in a reaction stronger than actually needed. To overcome this the designed flight-control first checks the read yaw-error. If the read error is less than -180 degrees a slip over the explained gap is detected and 360 degrees will be added to the calculated error, which would correct the earlier mentioned example to the wanted 5 degrees. Further it is needed to add the same behaviour for the other side: If the read error is bigger than 180 degrees, 360 degrees will be subtracted of it. On this way the gap will no longer be a problem.

4.3 The PID-Controller

As mentioned in Chapter 1 the flight-control measures the difference between the actual angles to the desired ones. Based on these errors it calculates how much throttle each rotor should have to get back to the desired position. These calculation is done by a so called PID-Controller. It consists of three parts: the *proportional*, the *integral* and the *derivative*. Each of these parts has a different task in the calculation process and all together provide the quadrocopter with fast reactions on disturbances such as wind or small collisions, but also with the ability to negate constant errors such as a miss placed mass centre.

4.3.1 The Parts P, I and D

In the following the different controller parts will be explained together with the formulas used to calculate their output. The given formulas contain axis-specific variables marked with the index X. To use these formulas on other axes it is necessary to use the variables specific for the used axis.

Proportional Controller

The first and most important part of the PID-Controller is the proportional one. It is a linear term that determines a reaction based on the current error. The following formula was used for the calculation:

$$P_X(n) = kP_X * E_X(n) \tag{1}$$

The value E_X describes the difference of the current angle on the x-axis to the desired one - thus the mentioned error. The value kP_X describes the so called *controller gain* of the p-controller on the x-axis. This gain changes the controllers behaviour. The smaller kP_X is chosen, the weaker the resulting reaction will be. The higher kP_X is chosen the stronger the resulting reaction will be. A value chosen too low could lead into a too slow reaction of the quadrocopter, which leads into a crash. A value chosen too high will lead into overshooting, which ends in an oscillating quadrocopter and finally into a crash. Thus it is very important to determine a proper value for this constant.

Integral Controller

Even if the p-controller is the core-element of the whole system it has a disadvantage: Without an error the controller does not react. So an only p-controlled flight-control would result into a quadrocopter with a steady error. This error creates a reaction that is strong enough to prevent the error from growing, but does not minimize it any further. The higher the chosen kP_X the smaller this steady error is. Nonetheless as mentioned in Section 4.3.1 this could lead into oscillating, thus eliminate the steady error just with the p-controller is not possible. For this task the integral part is needed. The errors get integrated over the time, which means that the longer the quadrocopter stays in an unwanted position the stronger the reaction of the i-controller will become. The used formula for this task is the following:

$$I_X(n) = I_X(n-1) + kI_X * E_X(n) * (t(n) - t(n-1)) \tag{2}$$

It can be seen that the i-controller has an controller gain kI_X too. This constant has the same problem as kP_X. A value chosen too low leads into a reaction not strong enough to correct errors and a value chosen too high can lead into overshooting. Both is not wanted so a determination of this value is needed too.

Derivative Controller

Both previous explained controller parts have the disadvantage that they only react on already existing errors. Desired is a stable flight, thus we do not want to get bigger error values. For this task the d-controller is added. This controller part reacts on a change of the error itself. The more the error changes between two readings, the stronger the reaction of the controller becomes. On this way the quadrocopter reacts already strong even if the error is still building up. The used formula for this is the following:

$$D_X(n) = kD_X * \frac{E_X(n) - E_X(n-1)}{t(n) - t(n-1)} \tag{3}$$

Like the two previous controllers the d-controller has a controller gain kD_X that also needs determination for a proper working controller.

4.3.2 Determine the Controller Constants

After the Controller itself is designed the next step is to determine the described controller gain constants. For this task there is no mathematically way, but a experimental way is described by Thomas Bräunl [Brä06]. The PID-Controller described by Bräunl is used to control the velocity of a system similar to a car. To adapt it on a quadrocopter some changes need to be done. In the beginning the values for the x- and y-axis need to be found. This can be done at the same time, because in a normal setup these two should have the same values. For this task the procedure consists of the following 5 steps:

1. Set all constants to 0 and begin with raising $kP_{X/Y}$, until the quadrocopter starts oscillating.

2. Divide the found value by 2.

3. Keep $kP_{X/Y}$ and raise $kD_{X/Y}$ until the quadrocopter starts oscillating.

4. Divide the found value by 2.

5. Test the flight of the quadrocopter. If it has an accelerating drift increase $kI_{X/Y}$.

The procedure to determine the constants for the z-axis works the same way, but it has to be performed separately, because the values will differ. It is important to say that before the i-constant for the x- and y-axis can be determined at least a p-value for the z-axis is needed. Without a controlling on the z-axis the quadrocopter might spin around, which makes it hard to observe the behaviour as needed for step 5.

After this process the controller-constant should be good enough to do small flight tests, but it is still possible to improve these values. To do so we added special commands for our bluetooth-communication to adjust these values while the system was running. With these enhancements we did tests, until a stable flight was reached. There are some simple rules how the constants should be tuned for a certain behaviour:

- ▻ If the quadrocopter reacts slowly on errors and starts to tumble raise $kP_{X/Y}$ and decrease $kD_{X/Y}$.
- ▻ If the quadrocopter drifts with an increasing velocity raise $kI_{X/Y}$.
- ▻ If the quadrocopter slowly starts oscillating at fast flights $kI_{X/Y}$ is too high or too low. This depends on the model.
- ▻ If the quadrocopter oscillates with a small amplitude on disturbances such as wind decrease $kD_{X/Y}$.
- ▻ If the quadrocopter reacts too slow on disturbances such as wind increase $kD_{X/Y}$ and maybe $kP_{X/Y}$.

At this point the flight-control itself is finished and the quadrocopter is able to perform a stable flight.

4.4 Manual Flight

As mentioned in Section 1.2 we included a possibility to remote control the quadrocopter via Bluetooth. To do so a simple list of control commands was developed. It is based on the principle of sending a character that is read by the flight-control to perform a certain action. The first design of this list is shown in Table 4.2.

To start the motors their throttle is set to a value, high enough to let them spin but too low to launch the quadrocopter. For the stop command the throttle is reduced to a point without movement. The reset command set the desired angles back to zero. The increase and decrease commands both change the throttle of each motor by one and the last six commands change the desired angle for the respective axis by one degree. The fact that we had chosen to use the quadrocopter indoors and thus interferences with other wireless communications such as Wireless LAN are possible was the reason to change this

Table 4.2. Unsafe Bluetooth commands

Character	Command
q	Start the motors
e	Stop the motors
r	Reset desired angles
w	Increase the throttle
s	Decrease the throttle
a	Turn counter clockwise
d	Turn clockwise
i	Tilt forward
k	Tilt backward
j	Tilt left
l	Tilt right

Table 4.3. Safe Bluetooth commands

Byte	Unsigned Integer Value	Command
00 00 11 11	15	Start the motors
00 11 00 11	51	Stop the motors
11 00 00 11	195	Reset desired Angles
00 11 11 00	60	Increase the throttle
11 00 11 00	204	Decrease the throttle
11 11 00 00	240	Turn counter clockwise
01 01 01 01	85	Turn clockwise
01 01 10 10	90	Tilt forward
01 10 01 10	102	Tilt backward
10 01 01 10	150	Tilt left
01 10 10 01	105	Tilt right

method. We wanted to make sure that one wrong transmitted bit can not lead into a wrong command. Thus we decided to send bytes with a *Hamming-Distance* of at least 4. This led to the final command list shown in Table 4.3.

4.5 Problems during Development

After the implementation and testing of the described flight-control we faced three problems:

1. Crash with Control Command
2. Corrupted Position Sensor Data
3. Initial Drift

In the following Subsections the reason of these problems will be explained and it will be described what was tried to solve these problems.

4.5.1 Crash with Control Command

At a certain point we observed the quadrocopter starting an extreme oscillating that needed an interruption of the flight test. After reviewing the debug data that were sent via the Bluetooth communication, it was seen that the calculated throttle-change in the moment of a control command was a value above 50, if the cycle time of the flight-control was as small as possible. The throttle range for the motors is a value between 0 and 255 and a minimum value of 150 is needed for the motors to run. Thus a change of over 50 would lead into two motors run at maximum speed and the other two would stop rotating. Even if this was just for one cycle the extreme reaction is enough to provoke a possible crash. Further researches showed that the problem in this case was the d-part of the PID-Controller. We had an error difference of $E_X(n) - E_X(n-1) = 1$ due to the control command. The small cycle time as mentioned was $t(n) - t(n-1) = 0.002s$. With the calculated controller gain of $kD_X = 0.11$ the resulting d-controller value following Formula 3 was $D_X(n) = 55$. To overcome this problem there were 4 different solutions:

1. *Ignore the d-value in case of control command.*
   This would be a possible solution but ignoring this part of the controller should be the last possibility, because it is actually needed for the quadrocopters stability and should work all the time.

2. *Smaller setpoint changes at control commands.*
   Reducing the amount of setpoint change at one control command would reduce the error difference that the d-controller value is depending on, but reducing the setpoint change would also result into a slower reaction of the quadrocopter on the control commands. The fact that the quadrocopter built in this project was meant to fly indoors means that a fast reaction possibility is needed, because of small rooms.

3. *Fix the cycle time on an amount higher than the longest measured one.*
   The other dependency of the d-controller value is the cycle time. Fixing it on a higher value would change the time in the calculation process of the d-controller from a variable into a constant. On this way the behaviour of the system would be more predictable but at the same time we reduce the reaction speed of the system.

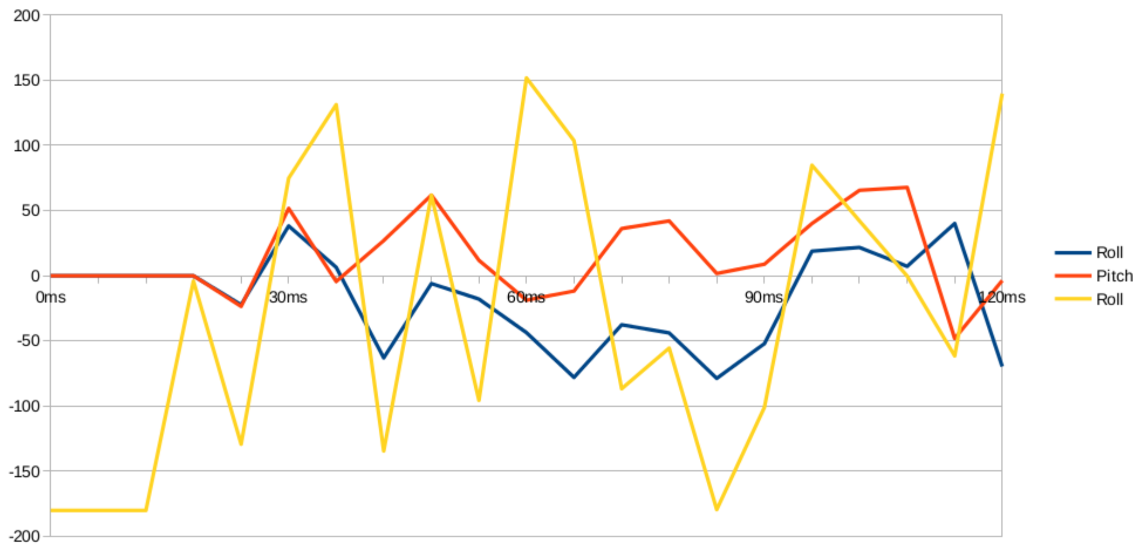

Figure 4.4. Random data received by the position sensor

4. *Spread the setpoint change over several cycles.*
   The idea is to stay with a setpoint change of 1 degree per command but spread this change over for example 5 cycles. On this way the error difference for the d-controller is smaller which leads into a smaller result, but the reaction of the quadrocopter on single control commands is barely changed.

We did not want to lose responding time of the system, thus we decided to test idea 4, the spread of setpoint change over several cycles, first. This change fixed the described problem and the other possible solutions were not tested.

4.5.2 Corrupted Position Sensor Data

After solving the problem with the control commands another problem was observed. During some flight tests the quadrocopter randomly started oscillating and sometimes even just stopped the motors without any user input. A review of the debug output showed that the position data received by the position sensor looked like random values, as shown in Figure 4.4.

The conclusion was that this data was corrupted in some way. The first thought was a damaged sensor, but the problem still existed after replacing the sensor. Further review showed that the problem was initialised by the package size of the position sensor data. The normal size of these packages is 42 bytes. Thus the reading algorithms given by the i2cDevLib are aligned to this size. Testing displayed that in the cycle were the mentioned phenomenon starts the package size is different. This leads to false readings. Due to the fact that the false package size is not constant but varies each time the fault occurs left no constant way to correct it. Thus the only possible solution was to clear the buffer of the position sensor and to discard the received data. On this way we created cycles without updated position data. Due to the fact, that the cycle time of the main controller is smaller than the one of the DMP, those cycles without new position data already existed earlier. These cycles would only become a problem if the appear with a high frequency. Further testing concluded that the more data the given debug output included the more often the mentioned error occurred. Thus we reduced the debug values as much as possible and reached a point were the error occurred with an average of 1 out of 10000 cycles.

4.5.3 Initial Drift

The last big problem we faced during the development of the flight-control was an initial drift. Every time the quadrocopter launched it drifted into one direction. The problem was that this drift was not accelerating but with constant velocity. This means the quadrocopter did perform a balanced flight but did receive an initial velocity during the launch process. This could occur for example by an unlevelled ground. The problem is that the built quadrocopter had no possibility to detect this velocity by itself. Thus the only way to work against this velocity was by manual controls. The idea to solve this problem was to use the accelerometer of the position sensor to integrate its values over time to calculate the quadrocopters velocities on all three axes. Nevertheless this idea had two problems: On the one hand the accelerometer is meant to detect the gravity. To use these data to calculate velocities the gravity needs to be extracted out of these values, which leads to unstable data. On the other hand an integration on a digital device always includes errors, which grow with every iteration of the process. An implementation of this idea confirmed the mentioned problems: The resulting velocities drifted. Even if the quadrocopter was not moving the calculated velocity on the x-axis was returned as $2\frac{cm}{s}$ after just 10 seconds and was still growing. Thus this was no proper solution and due to time reasons no other ideas had been tested.

Chapter 5

Implementation

In the following sections the two implementations of the in Chapter 4 explained flight-control will be described. Started with the one using C/C++.

5.1 Flight-Control Implementation using C/C++

This implementation consists out of 8 different files that are written either in Arduino code, which is very similar to C-Code and compatible with it, or in C/C++. The main file of this system is the so called *LibraryController.ino*. It is responsible to call all needed functions of the other parts and contains the setup and loop functions that are called by the Arduino. The setup-function as shown in Listing 5.1 initializes all needed parts of the system.

```
void setup() {

  motorInit();

  // initialize serial communication
  // (57600 chosen because it is the fastest Bautrate the Bluetoothsensor can work with)
  Serial.begin(57600);

  while (!Serial); // wait for Serialport to be established
  mpuSetup();


  //==============PID setup=====================================
  //Write control constants into controller objects

  pidRoll.ChangeParameters(PID_ROLL_KP, PID_ROLL_KI, PID_ROLL_KD, PID_ROLL_MIN, PID_ROLL_MAX);
  pidPitch.ChangeParameters(PID_PITCH_KP, PID_PITCH_KI, PID_PITCH_KD, PID_PITCH_MIN, PID_PITCH_MAX);
  pidYaw.ChangeParameters(PID_YAW_KP, PID_YAW_KI, PID_YAW_KD, PID_YAW_MIN, PID_YAW_MAX);
}
```

Listing 5.1. Setup function

Noticeable in this code fragment is that the values of the pid-constants are set for each controller. This is done to open the possibility to create different PID-Controller objects with different constants, what is needed, because at least the values for the yaw-controller differ from the other ones. This also gives the possibility to change these constants even if the system is running what was used as described in Section 4.3.2. After the setup-function finished the Arduino will constantly call the loop-function. Thus all things that need to be done during one cycle need to be called inside this function. The first thing is to read the data of the position sensor as shown in Listing 5.2.

```
 1 void loop() {
 2
 3   //Update Sensor values
 4   readMPU();
```

Listing 5.2. Begin of the loop function

After this a delay takes place that blocks the further functionality for at least the first 10 seconds. This is needed, because the position sensor has its own initialization process, which needs 3 to 10 seconds. During this time the read data is not usable, so we have to wait for this process to finish. This waiting is done by counting the amount of loop calls and wait one millisecond each time. If 10000 cycles were counted this setup-process is set finished and the resulting yaw-angle is set to be the further used offset to correct the zero point as explained in Section 4.2.3. This functionality is shown in Listing 5.3.

```
 6 //raise initialisation counter if necessary
 7 if (!setupDone) {
 8   m++;
 9   delay(1);
10 }
11
12  //finish initialisation if counter reached max value
13 if (!setupDone &&  m >= 10000) {
14   setupDone = true;
15   yawOffset = ypr[0];
16 }
```

Listing 5.3. Setup functionality inside the loop function

As soon as this initialization process is finished the actual flight-control is implemented as shown in Listing 5.4. It starts by correcting the yaw angle by the given offset and then reading the user input if there is one as shown in lines 20 to 23. Lines 26 to 31 are responsible for determining the current set angles based on the read user input from this or previous iterations as explained in Section 4.5.1. The next step done in the lines 34 to 39 is correcting the calculated error of the z-axis as explained in Section 4.2.3. Afterwards the determined errors are forwarded to the PID-Controller objects to determine the needed throttle-changes as shown in lines 42 to 44. These changes are used to calculate new motor values based on the in Table 4.1 described way. This takes place in lines 47 to 58.

```
if (setupDone) {
    //correct the Yaw angle to start at 0 degrees
    correctedYaw = ypr[0] - yawOffset;

    // Read incoming controlInputs
    readSerial();

    // Setting set values to controlInputs
    setY = controlInputY;
    setX = controlInputX;
    setZ = controlInputZ;

    //start the directioning process
    directionControl();

    //calculation to make sure the copter always corrects a yaw error over the shortest way
    yawError = correctedYaw - isZ;
    if (yawError < -180) {
      yawError += 360;
    } else if (yawError > 180) {
      yawError -= 360;
    }

    //calculate the controll values to provide a stable flight
    PIDroll_val = (int) pidRoll.Compute((float)ypr[2] - isX);
    PIDpitch_val = (int) pidPitch.Compute((float)ypr[1] - isY);
    PIDyaw_val = (int) pidYaw.Compute(yawError);

    //write the calculated values on the motors
    if (throttle >= 150) {
      m0_val = throttle + PIDroll_val + PIDpitch_val + PIDyaw_val;
      m1_val = throttle - PIDroll_val - PIDpitch_val + PIDyaw_val;
      m2_val = throttle - PIDroll_val + PIDpitch_val - PIDyaw_val;
      m3_val = throttle + PIDroll_val - PIDpitch_val - PIDyaw_val;
    } else {
      m0_val = throttle;
      m1_val = throttle;
      m2_val = throttle;
      m3_val = throttle;
    }
    motorWrite(m0_val, m1_val, m2_val, m3_val);
  }
```

Listing 5.4. Flight-control code

The calculated throttle-changes are only used if the base throttle value is greater or equal to 150. As explained in Section 4.5.1 the motors do not spin below this value, thus it is assumed that the quadrocopter is not flying at that point so a calculation is not needed.

The reading of the DMP data is very similar to the example given within the i2cDevLib. The main change is the amount of bytes read in one iteration as described in Section 4.5.2. Listing 5.5 shows in lines 1 to 6 the request for data. The array `fifoBuffer` has a size of 64 bytes, thus if the amount of bytes available at the DMP (as stored in the variable `fifoCount`) is greater than 64 not all data can be read. Further in lines 12 to 20 the received data is transformed into the needed angles by functions that a part if the i2cDevLib. This just happens if the amount of read bytes is a multiple of 42 to avoid the corrupted data as described in Section 4.5.2.

```
 1 // read a packet from FIFO
 2 if(fifoCount > 64) {
 3   mpu.getFIFOBytes(fifoBuffer, 64);
 4 } else {
 5   mpu.getFIFOBytes(fifoBuffer, fifoCount);
 6 }
 7
 8 // track FIFO count here in case there is > 1 packet available
 9 // (this lets us immediately read more without waiting for an interrupt)
10 fifoCount -= packetSize;
11
12 // display Euler angles in degrees
13 if(fifoCount % 42 == 0) {
14   mpu.dmpGetQuaternion(&q, fifoBuffer);
15   mpu.dmpGetGravity(&gravity, &q);
16   mpu.dmpGetYawPitchRoll(ypr, &q, &gravity);
17   ypr[0]= ypr[0] * 180/M_PI;
18   ypr[1]= ypr[1] * 180/M_PI;
19   ypr[2]= ypr[2] * 180/M_PI;
20 }
```

Listing 5.5. Reading the DMP data

The next step in the process is reading the user input, which is done by functions implemented in the file *CommunicationHandler.ino*. It is nothing more than reading the send byte if it is available and checking within a switch-statement whether one of the in Table 4.3 listed commands has arrived and then set the needed variables. Listing 5.6 shows an example of this for the tilt forward command.

```
case 90:
  controlInputX += 1;
  break;
```

Listing 5.6. Example of a movement command

The calculation of the new set angles is implemented as shown in Listing 5.7. Depending on whether there is a difference between the actual set angle and the desired one, the actual set angel is changed by the defined direction control step size. This is needed to spread the change of the set angles over several ticks as explained in Section 4.5.1.

```
void directionControl() {
  if(setY - isY != 0) {
  isY += directionControlStepSize * (setY - isY) / abs(setY - isY);
  }
  if(setX - isX != 0) {
    isX += directionControlStepSize * (setX - isX) / abs(setX - isX);
  }
  if(setZ - isZ != 0) {
    isZ += directionControlStepSize * (setZ - isZ) / abs(setZ - isZ);
  }
}
```

Listing 5.7. Calculation of new set angles

Finally the throttle-changes are calculated within the PID-class by using the in Section 4.3.1 described formulas as shown in Listing 5.8.

```
double PID::Compute(double mError) {
  //timestamp used for time depended control parts
  unsigned long tn = millis();
  double dt = (double)(tn - tp); //tp is the timestamp of previous iteration
  tp = tn;

  double P = (double) kp * mError;
  double D = (double) (kd * (mError - pError) * 1000.0 / dt); //pError is mError of previous
       iteration
  pError = mError;
  double I = (double) (Ip + ki * mError * dt / 1000.0); //Ip is calculated I value of previous
       Iteration
  Ip = I;

  double U = (double) (P + I + D);
  if (U > Hval) {
    U = Hval;
  } else if (U < Lval) {
    U = Lval;
  }
  return U;
}
```

Listing 5.8. Calculation of the throttle-change

5.2 Flight-Control Implementation using SCCharts

The same flight-control was also implemented with SCCharts as this was the main intention of this work. As explained in Section 1.3 an SCChart program takes inputs and calculates the corresponding outputs. In the case of the designed flight-control the needed inputs are the angular position, the user input and the current time stamp. The determination of these needs to be done in some kind of wrapper code that works the same way as it does for the C/C++ implementation. With these inputs the SCChart program calculates the needed motor values the same way as the C/C++ program as described in Section 5.1. As first impression a complete visual representation of the SCChart will be given in Figure 5.1. To provide a better readability the transition labels of this representation has been removed.

Noticeable about this is, that the program flow reaches a circle after passing the first two states. These first two states are used to do the initialization as described in Section 5.1. Another interesting point is, that all transitions inside the circle except one are immediate transitions. On this way it is guaranteed that the whole calculation process for the needed throttle-change is finished with one execution of the tick-function.

For a more detailed view onto the program the visual representation of the SCChart is divided into 5 parts and the transition labels are shortened. Figure 5.2 show, that after the mentioned initialization the state `correctYaw` is reached and the yaw offset is set. As known from the C/C++ implementation the next step is to correct the yaw angle, which is done as effect of the immediate transition that is leaving through the bottom of the picture. Another thing to notice is that this transition has no trigger, which means that every time we reach the state `correctYaw` the program will follow this transition, which is exactly what we want. The last state that can be seen in this figure is `motorValues`. This is the final state of the flight-control that is reached as soon as the throttle-change is calculated. The transition that leads from this `motorValues` to `correctYaw` is the only delayed transition in the flight-control process. On this way we make sure that every call of the tick-function will lead to this point, thus motor values are calculated on each iteration.

The next task that needs to be done is checking the user input. Figure 5.3 shows this process. The state `readCommand` is reached by the transition that enters on the bottom of the picture and comes from the state `correctYaw`. For every possible command following the list of Table 4.3 there is one transition with the value of the command as its trigger leaving `readCommand`. Next to them there is one transition without a trigger that has the lowest priority among them all. So even if there is no user input the control flow reaches the next state. If there is a user input the transition with the corresponding trigger will be chosen so the effect needed to execute the read command occurs. As example if the `tiltForwardCommand` is read a variable called `directionCounterX` is increased by a predefined `directionControlAmount`. These counter variables are used to provide the in Section 4.5.1 explained spread of set angle changes over several cycles. Based on these counter-variables the new set angles are determined by the next part of the program, where all 13 transitions

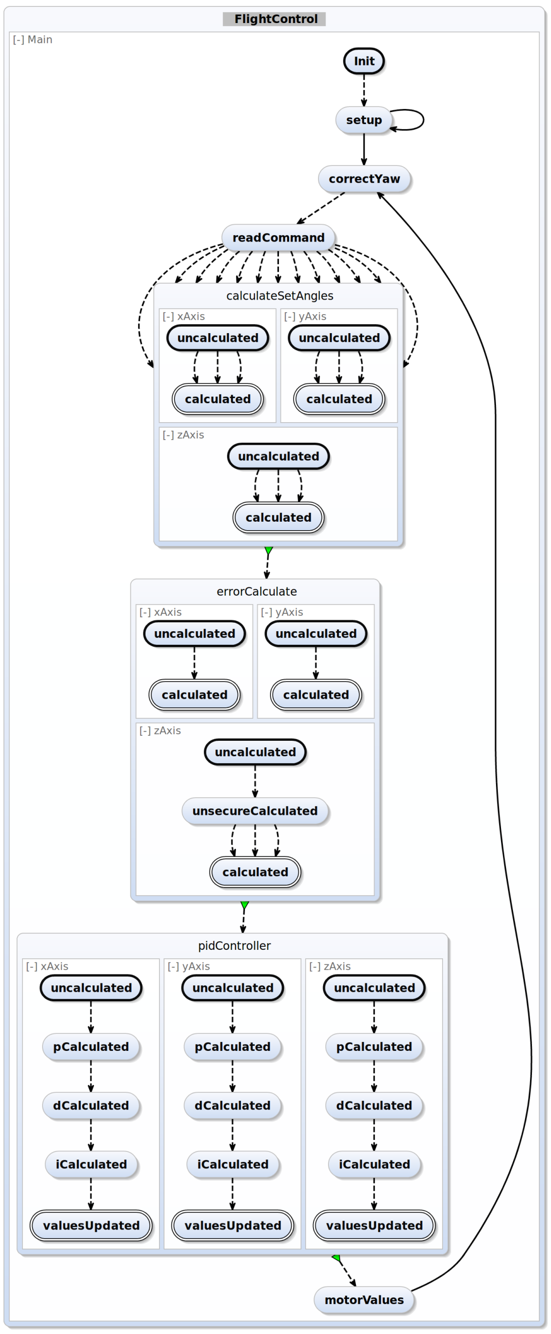

Figure 5.1. Complete SCChart without transition labels

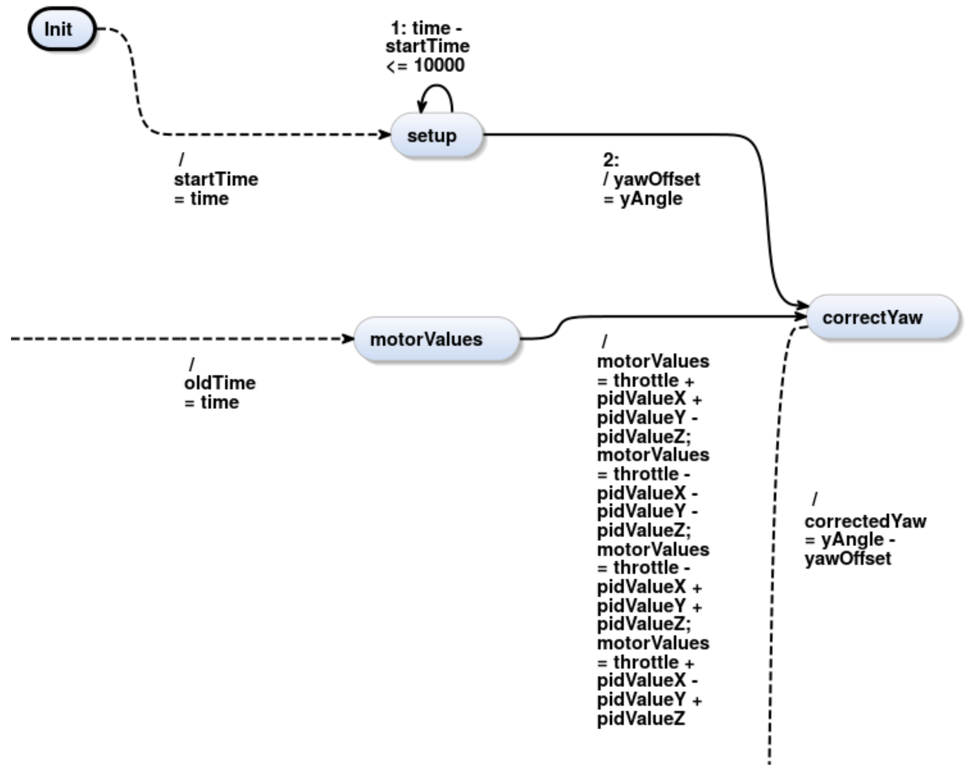
Init
1: time - startTime <= 10000
setup
/ startTime = time
2: / yawOffset = yAngle
correctYaw
motorValues
/ oldTime = time
/ motorValues = throttle + pidValueX + pidValueY - pidValueZ; motorValues = throttle - pidValueX - pidValueY - pidValueZ; motorValues = throttle - pidValueX + pidValueY + pidValueZ; motorValues = throttle + pidValueX - pidValueY + pidValueZ
/ correctedYaw = yAngle - yawOffset

Figure 5.2. SCChart initialization, yaw correction and writing the outputs

leaving `readCommand` are leading to.

The state reached after reading the user input is `calculateSetAngles`. Figure 5.4 shows these calculation exemplary for the x-axis. This state is a super-state that consists of three separate regions - one for each axis. In each of these regions the set angles for the representing axis are determined regarding to the counter values set by the user commands. If the corresponding counter variable has a positive value, this means that a raise of the set angle is needed. The effect of the transition does this by raising the set angle, depending on the chosen `directionControlAmount`. For every raise done by this way the counter variable is decreased by one, so the whole change for one user command is still 1 degree. The super-state is left by an termination transition without any trigger. Thus as soon as every region reaches its final state the super-state will be left.

The state reached next is called `errorCalculate` as shown in Figure 5.5. Like the state before this one is also a super-state of three regions that correspond to the three axis of the quadrocopter. These regions are used to determine the angular errors by calculating the difference between the current angle and the set angle and also to correct the error of the z-axis as described in Section 4.2.3 if needed. Like the state before, this state is left by a

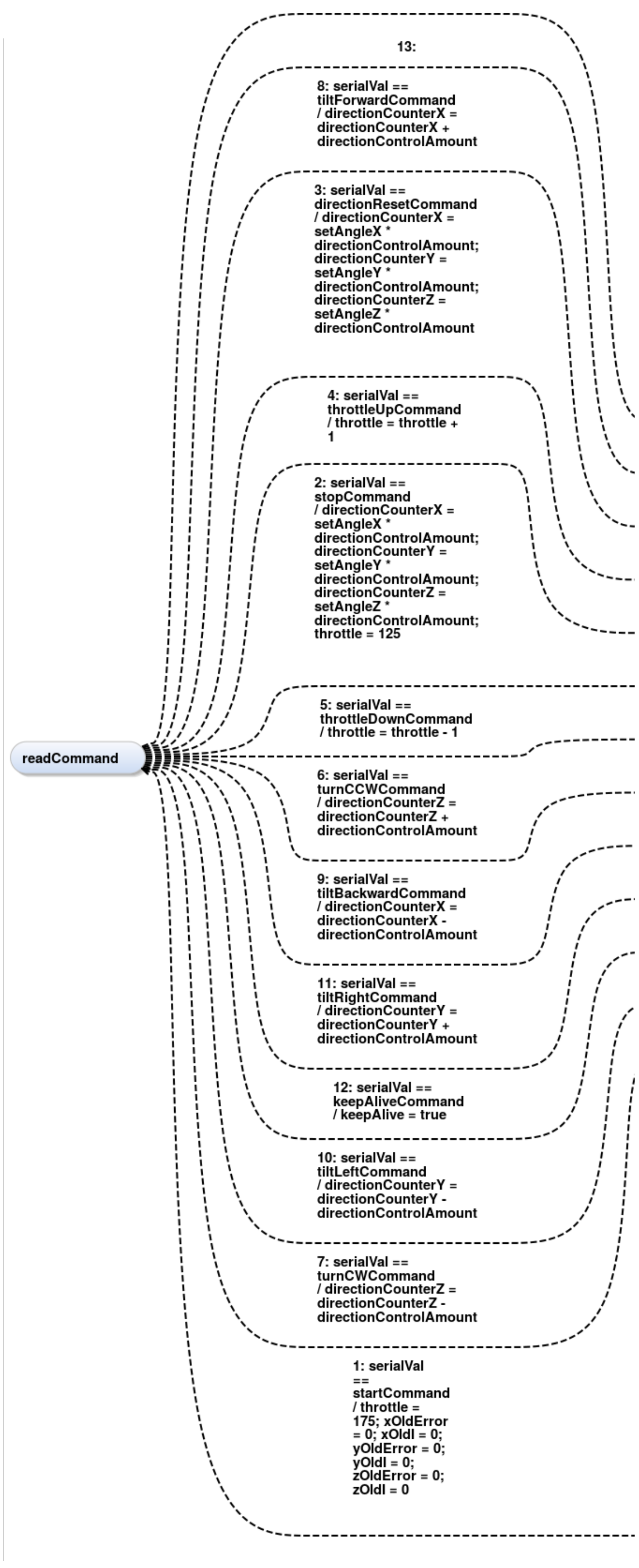
13:
8: serialVal == tiltForwardCommand / directionCounterX = directionCounterX + directionControlAmount
3: serialVal == directionResetCommand / directionCounterX = setAngleX * directionControlAmount; directionCounterY = setAngleY * directionControlAmount; directionCounterZ = setAngleZ * directionControlAmount
4: serialVal == throttleUpCommand / throttle = throttle + 1
2: serialVal == stopCommand / directionCounterX = setAngleX * directionControlAmount; directionCounterY = setAngleY * directionControlAmount; directionCounterZ = setAngleZ * directionControlAmount; throttle = 125
5: serialVal == throttleDownCommand / throttle = throttle - 1
readCommand
6: serialVal == turnCCWCommand / directionCounterZ = directionCounterZ + directionControlAmount
9: serialVal == tiltBackwardCommand / directionCounterX = directionCounterX - directionControlAmount
11: serialVal == tiltRightCommand / directionCounterY = directionCounterY + directionControlAmount
12: serialVal == keepAliveCommand / keepAlive = true
10: serialVal == tiltLeftCommand / directionCounterY = directionCounterY - directionControlAmount
7: serialVal == turnCWCommand / directionCounterZ = directionCounterZ - directionControlAmount
1: serialVal == startCommand / throttle = 175; xOldError = 0; xOldI = 0; yOldError = 0; yOldI = 0; zOldError = 0; zOldI = 0

Figure 5.3. SCChart user command read

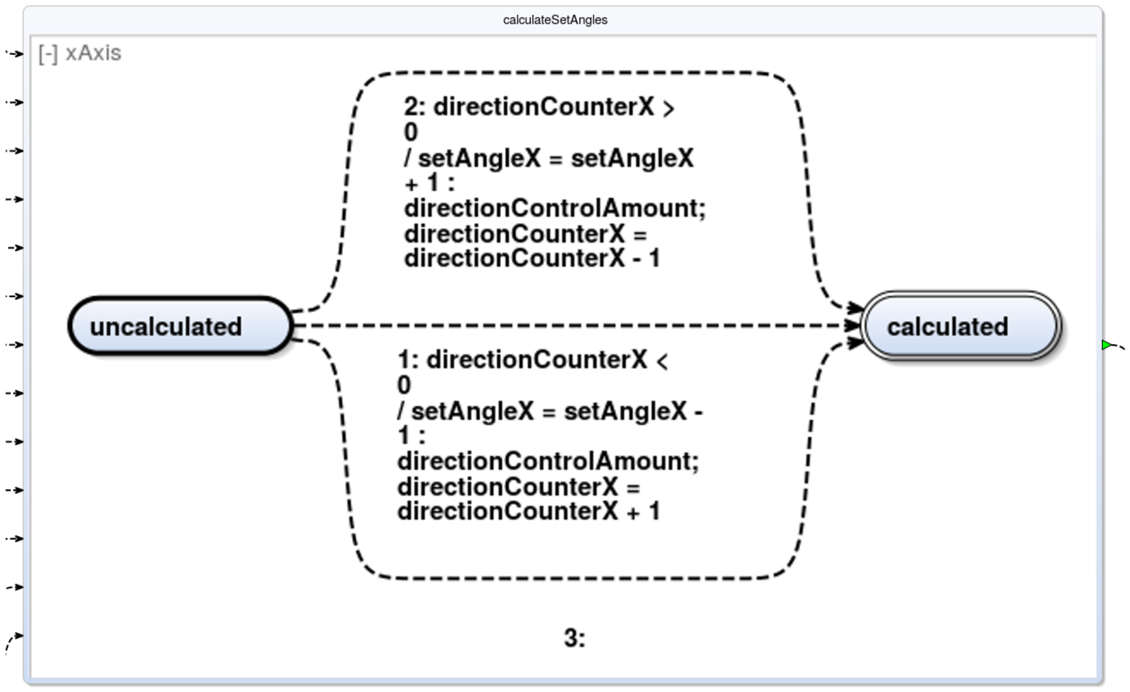
calculateSetAngles
[-] xAxis
uncalculated
calculated
2: directionCounterX > 0 / setAngleX = setAngleX + 1 : directionControlAmount; directionCounterX = directionCounterX - 1
1: directionCounterX < 0 / setAngleX = setAngleX - 1 : directionControlAmount; directionCounterX = directionCounterX + 1
3:

Figure 5.4. SCChart set angle calculation for the x-axis

termination transition for the same reason.

The last state of this SCChart is `pidController`. It is used to calculate the throttle-change for every three axes separately as shown for the z-axis in Figure 5.6. The process is separated into 4 different steps, one done by each transition inside a region. Each of these transitions has no trigger, so the corresponding effects will occur for each cycle. The first effect determined by the transition leading from `uncalculated` to `pCalculated` is used to calculate the reaction of the p-part of the PID-Controller. The following effect calculates the reaction of the d-part and after that the i-part is calculated. The last step for each region is to update the variables `OldI` and `OldError`, which is needed for the calculation of the next iteration. Also the sum of all controller parts is determined which is the resulting throttle-change. The calculation of all three controller parts follows like the C/C++ implementation the formulas given in Section 4.3.1. As soon as this is done the terminating transition of this state leads to `motorValues` shown in Figure 5.2 to write the motor values on the output and to end this cycle.

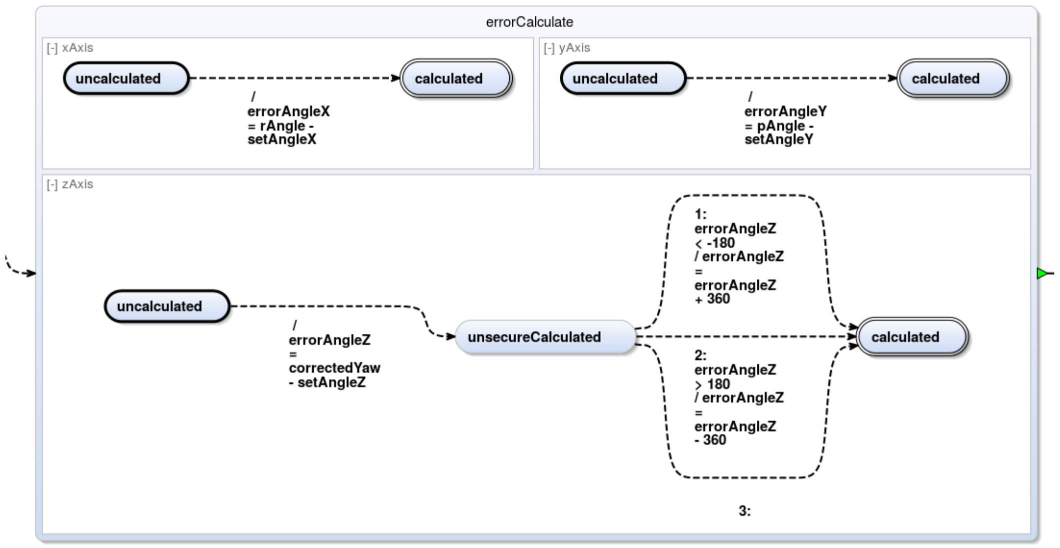
errorCalculate
[-] xAxis
uncalculated
calculated
/ errorAngleX = rAngle - setAngleX
[-] yAxis
uncalculated
calculated
/ errorAngleY = pAngle - setAngleY
[-] zAxis
uncalculated
/ errorAngleZ = correctedYaw - setAngleZ
unsecureCalculated
1: errorAngleZ < -180 / errorAngleZ = errorAngleZ + 360
2: errorAngleZ > 180 / errorAngleZ = errorAngleZ - 360
3:
calculated

Figure 5.5. SCChart error calculation

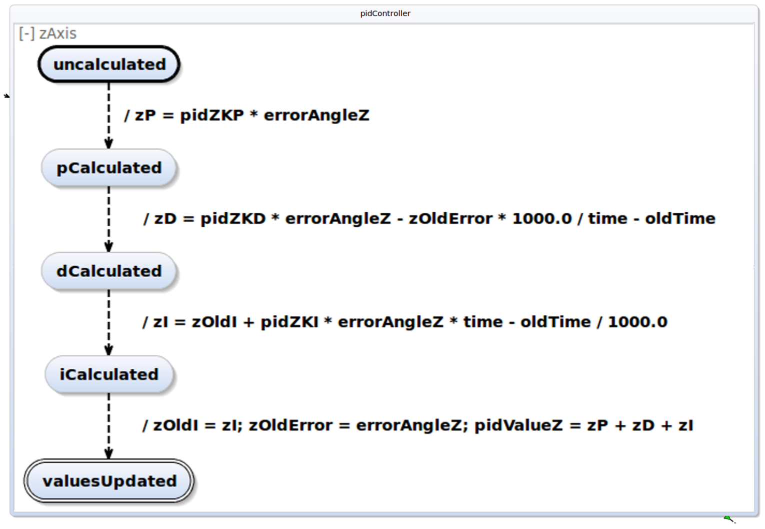
pidController
[-] zAxis
uncalculated
/ zP = pidZKP * errorAngleZ
pCalculated
/ zD = pidZKD * errorAngleZ - zOldError * 1000.0 / time - oldTime
dCalculated
/ zI = zOldI + pidZKI * errorAngleZ * time - oldTime / 1000.0
iCalculated
/ zOldI = zI; zOldError = errorAngleZ; pidValueZ = zP + zD + zI
valuesUpdated

Figure 5.6. SCChart PID-Controller for the z-axis

Chapter 6

Evaluation

To evaluate the work done, we will focus on a comparison of the SCChart implementation to the one done with C/C++. The comparison illustrates the benefits of using SCCharts or if it would be better to stay with the common approach. First we focus on the numbers:

C/C++ approach:
As mentioned in Section 5.1 the C/C++ implementation consists out of 8 different files excluding the used libraries and the code for obstacle avoidance. These 8 files consist of 758 lines of code and the whole system has an execution time of 1 to 6 milliseconds per cycle, which is depending on whether data is read from the DMP in this cycle or not.

SCChart approach:
The SCChart itself is just one file that consists of 36 states and 48 transitions. The KIELER-compiler translates it into 395 lines of c-code. To make this program run on the Arduino an extra of 219 lines of wrapper code is needed, which adds to a total of 614 lines of code excluding the used libraries and the obstacle avoidance code. The execution time of this system is 1 to 6 milliseconds per cycle depending on whether DMP data is read or not.

These numbers show that not just the execution time of both implementations is the same but also that the SCChart implementation is 144 lines shorter than the C/C++ implementation. This is surprising, because the translation of the SCChart results in automatically generated code which usually tends to be longer and sometimes slower in execution. Thus we see that for our project there is no execution related disadvantage in using SCCharts to implement a flight-control in comparison to C/C++. The benefit that comes with using SCCharts is the readability. As shown in Chapter 5 the graphically representation of SCCharts makes it easy to follow the flow of the program and the use of states and transitions provides a good overview about what is happening at which phase of the program. The disadvantage about working with SCCharts on this project was the necessity of wrapper code. Even if SCCharts provide a lot of functions and possibilities to model a system it is still needed to write extra code to read the sensor values and to write the calculated motor values onto the hardware. This means that it is not possible to implement the whole flight-control with SCCharts. Another point to mention would be that the implementation of a flight-control with SCCharts did not seem to be what

SCCharts was meant for: A system modelled with states and transitions usually has a conditional behaviour. This system stays in a state A, until a certain event X occurs. As soon as this happens the system switches to state B with a reaction Y. A flight-control on the other hand is a calculating system. It gets some inputs and based on them calculates some outputs. This means that a change of states is not needed to model a flight-control. Thus the whole system could have been implemented with one single state. The only advantage of using more states is readability.

The final conclusion of this evaluation is that SCCharts can be used to implement a fast working flight-control with great readability. Nevertheless the usage of other programming languages is necessary for completion.

Chapter 7

Conclusion

The final chapter of this thesis will be separated into two sections. Firstly a short summary will provide an overview about the work done and its results. Secondly an perspective about possible future work in this topic will be given.

7.1 Summary

The unmanned aerial vehicle quadrocopter needs the embedded system, called flight-control, to be able to fly. This system consists of two parts: A position sensor, that measures the angular position of the quadrocopter regarding to the earth level, and a PID-Controller, that calculates, based on the difference between the actual position to the desired one, how much the throttle of each motor has to be changed, so the quadrocopter can return to the desired position and stay stable on it. This flight-control can be implemented with different programming languages. It was shown that SCCharts is one of them. It was possible to implement a system that offers the same functionality as a C/C++ implementation with the same execution time and even less lines of code. This concluded that SCCharts is a tool as good as the common ones for the development of flight-control systems.

7.2 Future Work

Although the goal of designing a working flight-control for a quadrocopter was reached there are still possibilities to enhance this system. First of all the still existing problem of an initial drift that can only be corrected manually could be solved by using additional sensors to detect this drift and work against it. Furthermore the problem of corrupted DMP-data resulting of too much communication via the Bluetooth device could be investigated to remove this danger. Also the other suggested solutions for the problem described in Section 4.5.1 could be tested and compared to the used solution. Especially the use of a fixed cycle time could lead into a more stable and predictable system. Besides this one of the more distanced goals of the project was to create a fully autonomous quadrocopter. To reach this goal a navigation would be needed, because it is necessary to be able to state a route for an autonomous quadrocopter. Besides this the quadrocopter would be able to navigate itself to the declared target with such a navigation system. To reach this goal some kind of orientation points are needed, which the quadrocopter can detect to calculate its current position. The fact that the built quadrocopter was meant to fly indoors makes GPS unusable, but some sort of beacon technology could be a way to achieve this goal. Also a further extension of this work could be to integrate methods to detect system failures such as an empty battery or broken rotors and react on these errors as good as possible. On the side of SCCharts it would have been helpful if the recent integration of data-flow by A. Umland [Uml15] would be more elaborated as this would fit more for the implementation of a flight-control as described in Chapter 6. Further a possibility to avoid the needed wrapper code would be helpful to simplify the use of SCCharts for these kind of problems. The approach done by A. Stange [Sta15] could be a possibility to reach this goal.

Bibliography

[AH11] A. Zul Azfar and D. Hazry. "A simple approach on implementing imu sensor fusion in pid controller for stabilizing quadrotor flight control". In: *Signal Processing and its Applications* (2011), pp. 28–32.

[And03] C. André. Semantics of SyncCharts. Technical Report. ISRN I3S/RR-2003-24-FR. I3S Laboratory, Sophia-Antipolis, France, Apr. 2003.

[BDF+14] Klaus Bengler, Klaus Dietmayer, Berthold Färber, Markus Maurer, Christoph Stiller, and Hermann Winner. "Three decades of driver assistance systems". In: *IEEE Intelligent Transportation Systems Magazine* 6.4 (2014), pp. 6–22.

[BNS04] S. Bouabdallah, A. Noth, and R. Soegwart. "Pid vs lq control techniques applied to an indoor micro quadrotor". In: *Intelligent Robots and Systems* 3 (2004), pp. 2451–2456.

[Brä06] Thomas Bräunl. Embedded Robotics. Springer, 2006. ISBN: 3-540-34318-0.

[DAL12] Z. T. Dydek, A. M. Annaswamy, and E. Lavretsky. "Adaptive control of quadrotor uavs: a design trade study with flight evaluations". In: *Control Systems Technology* 21 (2012), pp. 1400–1406.

[HDM+13] Reinhard von Hanxleden, Björn Duderstadt, Christian Motika, Steven Smyth, Michael Mendler, Joaquín Aguado, Stephen Mercer, and Owen O'Brien. SC-Charts: Sequentially Constructive Statecharts for Safety-Critical Applications. Technical Report 1311. ISSN 2192-6247. Christian-Albrechts-Universität zu Kiel, Department of Computer Science, Dec. 2013.

[HDM+14] Reinhard von Hanxleden, Björn Duderstadt, Christian Motika, Steven Smyth, Michael Mendler, Joaquín Aguado, Stephen Mercer, and Owen O'Brien. "SC-Charts: Sequentially Constructive Statecharts for safety-critical applications". In: *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'14)*. Edinburgh, UK: ACM, June 2014.

[KWA+10] S. Klose, Jian Wand, M. Achtelik, G. Panin, F. Holzapfel, and A. Knoll. "Markerless, vision-assisted flight control of a quadrocopter". In: *Intelligent Robots and Systems* (2010), pp. 5712–5717.

[Mac15] Felix Machaczek. Collision Avoidance of Autonomous Safety-Critical Real-Time Systems. Bachelor Thesis. submitted. 2015.

[Pei15] Lars Peiler. Modeling Simulations of Autonomous, Safety-Critical Systems. Bachelor Thesis. submitted. 2015.

[Sta15] Andreas A. Stange. Comfortable SCCharts Modeling for Embedded Systems. Bachelor Thesis. submitted. 2015.

[Uml15] Axel Umland. "'Konzept zur Erweiterung von SCCharts um Datenfluss'". `http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/aum-dt.pdf`. Diploma thesis. Kiel University, Department of Computer Science, Mar. 2015.