

Project Quadcopter

Automotive Systems Master
Software Based Automotive Systems
ASM2-SB / SS 2010

Project Documentation



Prof. Dr. rer. nat. (Purdue Univ.) Dipl.-Ing. Jörg Friedrich
Tutor: Dionysios Satikidis

Joachim Schwab
Sreekanth Sundaresh
Nina Vetlugina
Marc Weber

General document information

A Quadrocopter is an aircraft with four, in single-plane arranged propellers, which use the thrust to achieve uplift, declination for propulsion and is able to start and land vertically. Due to the last characteristic, a quadrocopter can be classified as a helicopter.

The quadrocopter project was launched with the goal to build a system, which is developed completely by students of the University of Applied Sciences Esslingen. In a first project the Printed Circuit Board which hosts the parts necessary to control the quadrocopter was designed by students of the electronics elective in Göppingen. This project groups' focus was then on the simulation, implementation of the basic functions and visualization of the actual condition of the aircraft. The first two project groups already show the benefit of this project, a multidisciplinary development including hardware design, application development, embedded programming, simulation and the interfaces between these special fields. There are also enough possibilities for further enhancements like including a GPS-sensor for autonomous flights or a camera.

This document provides information about the actual state of the project. Chapter 1 starts by describing the system model for a quadrocopter and the calculation respectively the determination of variables needed. In chapter 2 at first the control algorithm, which is used in the current implementation, is presented followed by the sensor filter design, which provides information on how the signals from the sensors are processed to get the desired feedback signals. The same chapter also deals then with the transformation from the block diagram representation into the embedded version. After looking at the different parts of the quadrocopter in chapter 3, the embedded software with its general structure and its different layers are shown in chapter 4. To be able to adjust parameters, used in control algorithms or in other software, there is a wireless connection to a base station available. The communication protocol is described in chapter 5. Finally, chapter 6 presents the realization of the base station software. The document ends with a short summary.

Files and programs

For practical reasons all files (models, parameter files and source code), which are mentioned in this documentation, can be found in a subfolder of this report and not as an appendix.

MATLAB/Simulink® 2009b was used as simulation environment. Unfortunately the software is not available as free download. But students of the HS Esslingen have access to the software in all PC-pools.

For the development of the embedded software, the cross-compiler Freescale CodeWarrior for HCS12 V5.0 Special Edition (code size limitation) was used. It can be downloaded from the Freescale website – www.freescale.com.

The BDM programmer of PEmicro is used to flash the microcontroller. The required drivers (ver. 10) can be downloaded from the website of PEmicro – www.pemicro.com.

The XBee Development Kit of the Digi company was used for development and testing of the wireless communication between quadrocopter and base station. This includes a program called X-CTU (ver. 5.1.4.1) with which the wireless communication can be observed. The program itself and the corresponding drivers are available for download on the Digi website – www.digi.com.

NetBeans 6.9 was used as integrated development environment (IDE) for the base station software, which is written in Java. The IDE can be downloaded from www.netbeans.org.

Table of contents

General document information	I
Files and programs	II
Table of contents	III
Equation index.....	VI
Figure index.....	VII
Table index	IX
1 Quadrocopter System Model	1
1.1 Basics.....	1
1.2 Equations of motion.....	5
1.2.1 Coordinate systems	5
1.2.2 Theory	6
1.1.1 Implementation	11
1.2.3 Determination of the moment of inertia	14
1.2.4 Basic elements.....	14
1.2.5 Steiner's parallel-axis theorem.....	16
1.2.6 Inertia of the simplified quadrocopter model.....	17
1.2 Determination of the thrust-set point curve	24
1.3 Improvements and conclusion.....	26
2 Quadrocopter Control Algorithm and Sensor Filtering.....	27
2.1 Control Algorithm	27
2.2 Control Modelling	28
2.3 PID Techniques.....	31
2.3.1 Roll Control	33
2.3.2 Pitch Control.....	34
2.3.3 Yaw Rate Control	34
2.4 Sensor Filtering.....	36
2.4.1 Complementary Filter	36
2.4.2 Calculation of the angle from x-axis accelerometer value	37
2.5 Model Transformation into Embedded Version Realization.....	41
2.5.1 Brief Info about S-Functions	41
2.5.2 Simulation Stages and S-Function Routines	42
2.6 Implemented S-Function blocks of Quadrocopter.....	44
2.6.1 S-Function block Quadrocopter Controller.....	44
2.6.2 S-Function block Sensor Filter.....	45
2.7 Visual Studio settings for compiling S-Functions.....	46
2.8 Test cases & Simulation results.....	53

2.9	<i>Improvements & Conclusions</i>	55
3	Quadcopter Electronics	56
3.1	<i>System Description</i>	56
3.2	<i>Central Control Unit</i>	57
3.3	<i>Accelerometer</i>	59
3.4	<i>Gyroscopes / Temperature Sensor</i>	60
3.5	<i>Battery Sensor</i>	60
3.6	<i>Remote Control</i>	60
3.7	<i>XBee Pro</i>	62
3.8	<i>Brushless Controller</i>	62
3.9	<i>Brushless Motor</i>	65
3.10	<i>Beeper</i>	65
3.11	<i>Light Emitting Diodes (LEDs)</i>	65
3.12	<i>Other Components</i>	66
4	Embedded Software	67
4.1	<i>General Structure</i>	67
4.2	<i>Hardware Abstraction Layer (HAL)</i>	69
4.2.1	<i>QH_accelerometer</i>	69
4.2.2	<i>QH_atd (Analog to Digital Converter)</i>	70
4.2.3	<i>QH_beeper</i>	71
4.2.4	<i>QH_brushless</i>	71
4.2.5	<i>QH_eeprom</i>	72
4.2.6	<i>QH_led</i>	73
4.2.7	<i>QH_pll</i>	73
4.2.8	<i>QH_remote</i>	73
4.2.9	<i>QH_timer</i>	75
4.2.10	<i>QH_xbee</i>	76
4.3	<i>Data Layer</i>	77
4.3.1	<i>Calibration parameter (CopterConfig)</i>	78
4.3.2	<i>System states (CopterState)</i>	79
4.3.3	<i>copter</i>	80
4.4	<i>Application Layer</i>	81
4.4.1	<i>main</i>	83
4.4.2	<i>basestation</i>	83
4.4.3	<i>flightcontrol</i>	83
4.4.4	<i>sensorfilter</i>	84
4.5	<i>Conclusion and further improvements</i>	84
5	Quadcopter – Base Station Communication	86
5.1	<i>General Information</i>	86
5.2	<i>Basic message layout</i>	86
5.3	<i>Physical data transmission</i>	87

5.4	Parameter Request (base station → quadrocopter)	88
5.5	Parameter Response (quadrocopter → base station)	88
5.6	Parameter Update (base station → quadrocopter)	88
5.7	Status Request (base station → quadrocopter)	89
5.8	Status Response (quadrocopter → base station)	89
6	Basestation Software	90
6.1	Installation guide	90
6.1.1	Installation Com port drivers	90
6.1.2	Installation of libraries for JFreeChart	91
6.1.3	Installation X-CTU Software & USB drivers	93
6.2	Software description	94
6.2.1	class PortSettings	95
6.2.2	class BytesTransformation	97
6.2.3	public class Objects	99
6.2.4	public class SimpleRead_response	99
6.2.5	public class Grafics	100
6.2.6	public class Setting	101
6.2.7	public class SimpleRead_parametr_response_vse	101
6.2.8	public class Utils	103
6.2.9	public class PropMgr	103
6.2.10	public class crc2	104
6.2.11	public class readfromfile_mod()	104
6.2.12	public class writetofile	105
6.2.13	public class RequestChooser()	105
6.3	User Interface	107
6.3.1	Status Request	107
6.3.2	Status Response	108
6.3.3	Parameter request	109
6.3.4	Parameter Update	110
6.3.5	Parameter Response	110
6.3.6	Chooser	111
6.4	Summary	112
7	Conclusion	113
8	Appendix	114
8.1	Weight of Frame Parts	114
8.2	Schematic of the central control unit	116
8.3	Subsystem Interfaces	117
8.3.1	Subsystem Quadrocopter Controller	117
8.3.2	Subsystem Sensor Filter	120
	References	122

Equation index

Equation 1.1: Kinematics of 6 DOF rigid body	6
Equation 1.2: Position vector in earth-fixed frame	6
Equation 1.3: Velocity vector in body-fixed frame	6
Equation 1.4: Rotation matrix	7
Equation 1.5: Examples how to build equations of motion	7
Equation 1.6: State vector of hybrid frame	8
Equation 1.7: Equations of motion in matrix form	8
Equation 1.8: Input values for quadcopter model	9
Equation 1.9: Gravitational force	9
Equation 1.10: Gyroscopic effects	10
Equation 1.11: Total input vector	10
Equation 1.12: Equations of motion ($c_\psi = \cos\Psi$, $s_\psi = \sin\Psi$)	10
Equation 1.13: Diagonal inertia matrix	14
Equation 1.14: Inertia of cuboid	14
Equation 1.15: Inertia of hemisphere	15
Equation 1.16: Inertia of cylinder	16
Equation 1.17: Steiner's parallel-axis theorem	16
Equation 1.18: Inertia of the electronics box	18
Equation 1.19: Inertia of the battery pack	19
Equation 1.20: Inertia of boom one	20
Equation 1.21: Symmetry conditions for booms	20
Equation 1.22: Inertia of motor one	21
Equation 1.23: Symmetry conditions for motors	21
Equation 1.24: Inertia of rotor one	22
Equation 1.25: Symmetry conditions for rotors	22
Equation 1.26: Inertia of rotor around rotors center of mass	23
Equation 1.27: Complete inertia of quadcopter	23
Equation 2.1: Equations of Motion	28
Equation 2.2: Basic movements versus individual propellers forces	28
Equation 2.3: Simplified & Inverted Equations of motion	29
Equation 2.4: Alternate form of Simplified & Inverted Equations of motion	30
Equation 2.5: Equations for forces of individual propellers	31
Equation 2.6: Basic PID equation in Time domain	32
Equation 2.7: Complementary Filter Equation	38
Equation 2.8: Numerical integration part of the equation	39
Equation 2.9: Low-Pass filter part of the equation	39
Equation 2.10: High-Pass filter part of the equation	39
Equation 2.11: Time constant of the filter	40

Figure index

Figure 1.1: Abstraction of Quadcopter	1
Figure 1.2: Throttle movement.....	2
Figure 1.3: Roll movement	3
Figure 1.4: Pitch movement.....	4
Figure 1.5: Yaw movement.....	4
Figure 1.6: Used coordinate systems	5
Figure 1.7: Implementation of the quadcopter model	11
Figure 1.8: Processing of model input data	12
Figure 1.9: Inertia of cuboid	15
Figure 1.10: Inertia of hemisphere	15
Figure 1.11: Inertia of cylinder.....	16
Figure 1.12: Steiner's parallel-axis theorem.....	17
Figure 1.13: Simplified model of quadcopter	17
Figure 1.14: Inertia of the electronics box	18
Figure 1.15: Inertia of the battery pack.....	19
Figure 1.16: Inertia of boom one.....	20
Figure 1.17: Inertia of motor one	21
Figure 1.18: Inertia of rotor one.....	22
Figure 1.19: Thrust to set point curve	24
Figure 1.20: Speed to thrust curve	25
Figure 2.1: Control Block Diagram	30
Figure 2.2: PID Structure	32
Figure 2.3: Block diagram of the Roll Control	33
Figure 2.4: Block diagram of the Pitch Control.....	34
Figure 2.5: Block diagram of the Yaw Rate Control	34
Figure 2.6: Quadcopter lying on the reference platform	37
Figure 2.7: Quadcopter tilted w.r.t. the reference platform	37
Figure 2.8: Fusion of Accelerometer & Gyro to get the filtered angle estimate	38
Figure 2.9: How Simulink® Performs Simulation	42
Figure 2.10: S-Function Quadcopter controller	44
Figure 2.11: S-Function Sensor Filter	45
Figure 3.1: Electronic system of the quadcopter	56
Figure 3.2: PCB of the central control unit	58
Figure 3.3: Hardware interfaces	58
Figure 3.4: SPI principle.....	59
Figure 3.5: Remote control receiver with sum-signal output.....	61
Figure 3.6: PPM sum-signal frame	61
Figure 3.7: XBee Pro modul	62
Figure 3.8: Brushless controller.....	63
Figure 3.9: Solder jumper on brushless controller PCB to select the address	63
Figure 3.10: Transmission of new setpoint to a brushless controller	64
Figure 3.11: Robbe Roxxy brushless motor	65

Figure 3.12: BDM programmer from PEmicro	66
Figure 4.1: Structure of the embedded software.....	67
Figure 4.2: Double buffering of remote values.....	74
Figure 4.3: Bits indicating if a time interval has elapsed	75
Figure 4.4: XBee double buffering and buffer exchange block mechanism.....	76
Figure 4.5: Software modules, accessing the data structures	77
Figure 4.6: Program flow in principle.....	82
Figure 5.1: General message layout.....	87
Figure 6.1: The Library Manager.....	92
Figure 6.2: Class interconnection tree	94
Figure 8.1: Schematic of the central control unit	116
Figure 8.2: Subsystem Quadrocopter Controller	117
Figure 8.3: Subsystem Sensor Filter	120

Table index

Table 3.1: Brushless controller addresses	64
Table 4.1: Data types	68
Table 4.2: Brushless controller addresses	71
Table 4.3: Calibration parameter	78
Table 4.4: System states.....	80
Table 6.1: Java primitive's data types	97
Table 6.2: The two's complement	98
Table 8.1: Weight of quadrocopter parts.....	115
Table 8.2: Inputs for Subsystem Quadrocopter Controller.....	118
Table 8.3: Outputs for Subsystem Quadrocopter Controller.....	118
Table 8.4: Parameters for Subsystem Quadrocopter Controller.....	119
Table 8.5: Constants for Subsystem Quadrocopter Controller.....	119
Table 8.6: Inputs for Subsystem Sensor Filter	120
Table 8.7: Outputs for Subsystem Sensor Filter.....	120
Table 8.8: Parameters for Subsystem Sensor Filter	121
Table 8.9: Constants for Subsystem Sensor Filter	121

1 Quadcopter System Model

1.1 Basics

In this section the basic information for understanding the behavior of a quadcopter is explained. The basic structure is given by a crosswise arrangement of four motors. In the middle of this cross all the other components, like the battery pack and the PCBs are located. Although these parts are responsible for the ability of the quadcopter to fly, their functions are not necessary to explain the basic performance and will be treated in the other chapters.

As already mentioned, the quadcopter consists of four motors, which are arranged crosswise. Two of them (front and rear) are rotating clockwise; the other two (left and right) rotate anti-clockwise. The rotors do also have numbers from 1 to 4, starting with the front motor as number 1 and incrementing the numbers anti-clockwise. The origin of the body-fixed coordinate system is located in the center of gravity of the whole structure. The x-axis points to the front of the quadcopter, the y-axis to the left and the z-axis to the sky in order to build a right-handed coordinate system as seen in Figure 1.1.

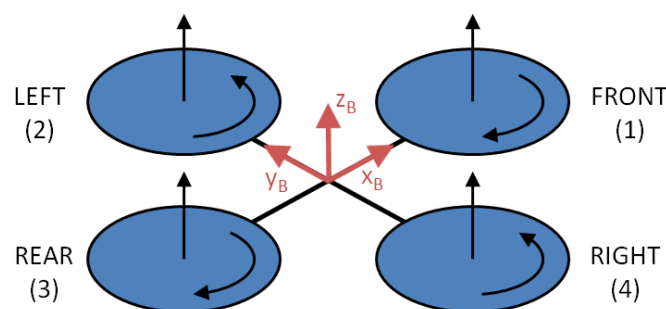


Figure 1.1: Abstraction of Quadcopter

The quadcopter has six degrees of freedom (DOF), which are split up in linear movements (x-, y- and z-direction) and angular movements (roll, pitch and yaw), but only four actuators which means that only set points for four DOF can be achieved directly. The other two have to be met by a combination of two DOF. In the following paragraphs these directly controllable DOF are described, with the assumption that ω is the speed where the quadcopter is in horizontal plane.

- Thrust

For accelerating in z-direction of the body-fixed coordinate system, all rotors have to increase their speeds by the same amount to achieve a higher thrust, which is then bigger than the gravitational force of the quadcopter and results in acceleration. If the attitude of the quadcopter is not horizontal, increasing the speed of the rotors will not only lead to a vertical movement but also to a horizontal movement.

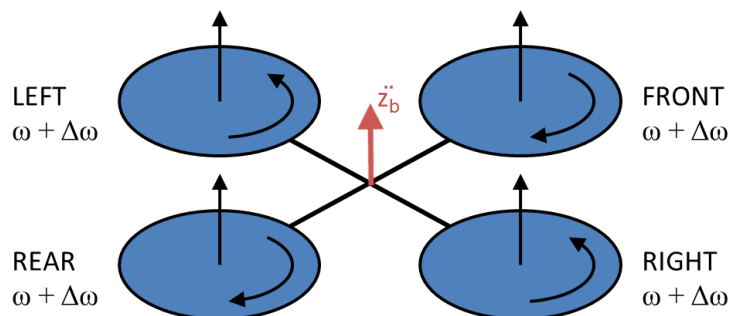


Figure 1.2: Throttle movement

- Roll

Rolling describes a change of the angle around the x-axis. This movement, together with additional thrust allows the quadrocopter to move in y-direction (one of the missing DOF). It can be achieved by changing the speed of the left and right rotor resulting in a torque around the x-axis. The value for $\Delta\omega$ should be added and subtracted by the same amount at the left and right rotor because otherwise a secondary movement will occur around the z-axis. The cause of this effect will be discussed in the description of the forth DOF (Yaw). By definition a clockwise rotation around the x-axis, looking to positive values of x, is to be seen as a positive angle.

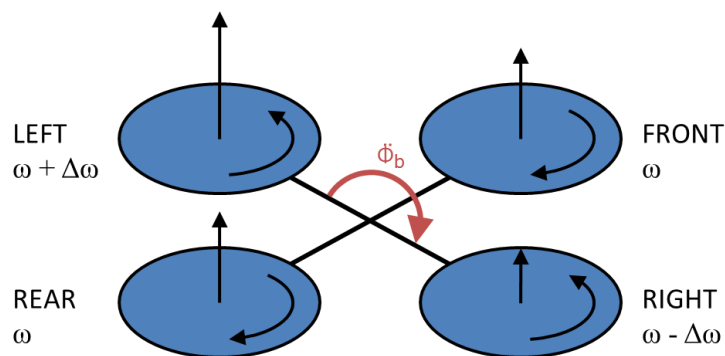


Figure 1.3: Roll movement

- Pitch

A Pitch angle describes an angle around the y-axis. Together with additional thrust a movement in x-direction can be realized. Reducing (or increasing) the speed of the front rotor and increasing (or decreasing) the speed of the rear rotor leads to a positive (or negative) torque and a positive (or negative) pitch angle around the y-axis. Again, the value of $\Delta\omega$ has to be the same for both rotors, otherwise it would not be a pure rotation around the y-axis.

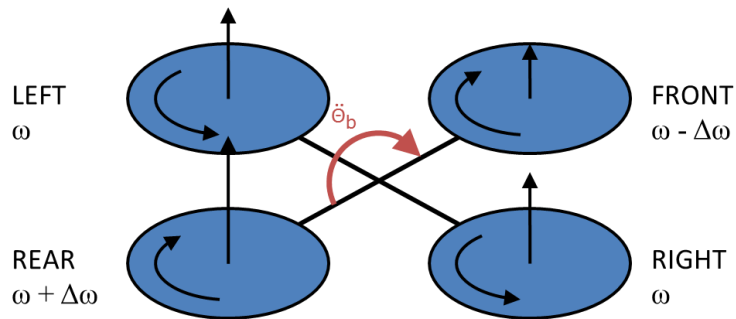


Figure 1.4: Pitch movement

- Yaw

Yawing describes the movement of the quadrocopter around the z-axis. For this type of behavior not the forces are of importance but another effect resulting from the rotation of the rotors has to be considered. With the rotation of the rotors, not only a force in z-direction is generated but also a drag torque which always acts to the opposite direction of rotation. From this follows that an increased (or decreased) speed of the left/right rotor couple and a decreased (or increased) speed of the front/rear rotor couple will lead to an positive (or negative) torque and therefore acceleration around the z-axis.

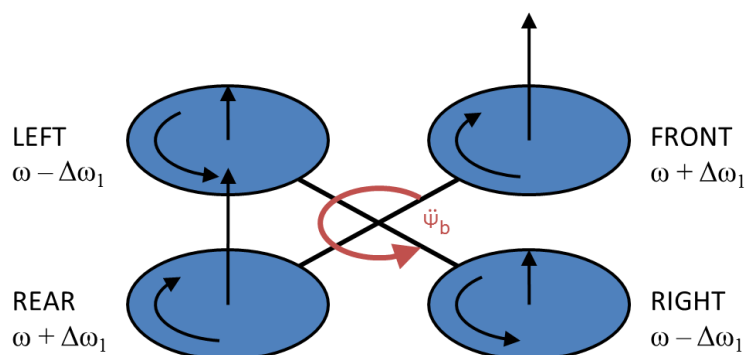


Figure 1.5: Yaw movement

1.2 Equations of motion

The behavior of the quadcopter can be described by differential equations. Using these equations it is possible to develop a model in a simulation environment like MATLAB/Simulink®. In this chapter the equations are not derived in very much detail. For further information see (Bresciani, 2008).

1.2.1 Coordinate systems

In the previous section the body-fixed coordinate system was already introduced. This system has its origin in the center of mass of the quadcopter with the x-axis directed to the front motor, the y-axis looking to the left motor and the z-axis pointing to the sky.

For describing the movement of the quadcopter there is also another coordinate system necessary which is called the earth-fixed coordinate system. Whereas the body-fixed system moves and rotates with the quadcopter, the earth-fixed system is in a fixed position on earth. X- and y-axis are describing the horizontal plane and the z-axis is pointing to the sky. The orientation of x- and y- direction is in the beginning identical to the direction of the body-fixed x- and y-direction. To distinguish both coordinate systems an index is written behind the letter. An e stands for the earth-fixed coordinate system and a b stands for the body-fixed coordinate system, see Figure 1.6.

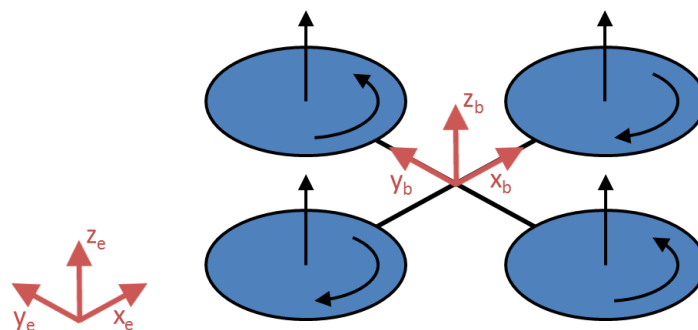


Figure 1.6: Used coordinate systems

1.2.2 Theory

The kinematics of a 6 DOF rigid body are given by

$$\dot{\xi} = J_{\theta} v$$

Equation 1.1: Kinematics of 6 DOF rigid body

Whereas $\dot{\xi}$ represents the velocity vector in the earth-fixed coordinate system, v is the velocity vector in the body-fixed frame and J_{θ} is the rotation and transfer matrix.

The position vector of a 6 DOF model with respect to the earth-fixed coordinate system consists of linear Γ^e and angular positions Θ^e as shown in Equation 1.2

$$\xi = [\Gamma^e \quad \Theta^e]^T = [X \quad Y \quad Z \quad \varphi \quad \theta \quad \psi]^T$$

Equation 1.2: Position vector in earth-fixed frame

The state vector with respect to the body-fixed coordinate system consists of linear velocities and angular velocities.

$$v = [V^b \quad \omega^b]^T = [u \quad v \quad w \quad p \quad q \quad r]^T$$

Equation 1.3: Velocity vector in body-fixed frame

For the transformation of v into $\dot{\xi}$ the matrix

$$J_{\theta} = \begin{bmatrix} R_{\theta} & 0 \\ 0 & T_{\theta} \end{bmatrix} = \begin{bmatrix} c_{\psi}c_{\theta} & -s_{\psi}c_{\theta} + c_{\psi}s_{\theta}s_{\varphi} & s_{\psi}s_{\theta} + c_{\psi}s_{\theta}c_{\varphi} & 0 & 0 & 0 \\ s_{\psi}c_{\theta} & c_{\psi}c_{\theta} + s_{\psi}s_{\theta}s_{\varphi} & -c_{\psi}s_{\theta} + s_{\psi}s_{\theta}c_{\varphi} & 0 & 0 & 0 \\ -s_{\theta} & c_{\theta}s_{\varphi} & c_{\theta}c_{\varphi} & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & s_{\varphi}t_{\theta} & c_{\varphi}t_{\theta} \\ 0 & 0 & 0 & 0 & c_{\varphi} & -s_{\varphi} \\ 0 & 0 & 0 & 0 & \frac{s_{\varphi}}{c_{\theta}} & \frac{c_{\varphi}}{c_{\theta}} \end{bmatrix}$$

Equation 1.4: Rotation matrix

with R_{θ} as the rotation matrix and T_{θ} as the transfer matrix is used.

Out of Equation 1.2 follows that six equations have to be created. The first three equations are responsible for describing the linear movement in x-, y- and z-direction with respect to the earth-fixed coordinate system. With the last three equations the angular motion around the x-, y- and z-axis with respect to the body-fixed coordinate system is described. The frame which consists of the combination of two different coordinate systems is called hybrid frame h. See Equation 1.5 for examples how equations for linear and angular accelerations are built.

$$\begin{aligned} m\ddot{\Gamma}^e &= \sum \text{all forces which attack in x-direction} \\ \ddot{\Gamma}^e &= \frac{1}{m} \sum \text{all forces which attack in specific axis} \\ J\ddot{\Theta}^b &= \sum \text{all torques which act around specific axis} \\ \ddot{\Theta}^b &= \frac{1}{J} \sum \text{all torques which act around specific axis} \end{aligned}$$

Equation 1.5: Examples how to build equations of motion

The used state vector with respect to the hybrid frame therefore looks like in Equation 1.6.

$$\zeta^h = [\dot{r}^e \quad \omega^b]^T$$

Equation 1.6: State vector of hybrid frame

Representing the equations of motion can also be done in a matrix form. According to (Bouabdallah, 2007) the equations of motion for a 6 DOF model can be seen in Equation 1.7 where I represents the inertia matrix and $I_{3 \times 3}$ is a 3 by 3 identity matrix.

$$\begin{bmatrix} m \cdot I_{3 \times 3} & 0_{3 \times 3} \\ 0_{3 \times 3} & I \end{bmatrix} \begin{bmatrix} \ddot{r}^e \\ \dot{\omega}^b \end{bmatrix} + \begin{bmatrix} 0_{3 \times 1} \\ \omega^b \times (I \cdot \omega^b) \end{bmatrix} = \begin{bmatrix} F^e \\ \tau^b \end{bmatrix}$$

Equation 1.7: Equations of motion in matrix form

Additionally to the equations above there is a term which takes the Coriolis Effect into account. This effect occurs when actions are observed from a rotating reference frame. As an example, consider two kids on a carousel throwing a ball to each other. From the perspective of the kids, the ball will fly in a curved path. This is caused by the Coriolis Effect. Because the path of an object changes without a visible force, they are also called fictitious forces.

Using the approach of Equation 1.7 there have also been two assumptions been taken into account:

- The center of mass is located in the same point as the origin of the body-fixed coordinate system.
- The axes of the body-fixed coordinate system coincide with the main axes of inertia. This allows using a diagonal matrix of inertia.

The input vector on the right side of Equation 1.7 does not only consist of the input forces and torques of the rotors but there are also other effects which have to be respected.

The biggest influences to the model do have the input forces and torques of the turning rotors. This input vector shall be called U with its components U_1 , U_2 , U_3 and U_4 .

$$U_h = \begin{bmatrix} R_\Theta & 0_{3 \times 3} \\ 0_{3 \times 3} & I_{3 \times 3} \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ U_1 \\ U_2 \\ U_3 \\ U_4 \end{bmatrix} = \begin{bmatrix} (s_\psi s_\varphi + c_\psi s_\theta c_\varphi) U_1 \\ (-c_\psi s_\varphi + s_\psi s_\theta c_\varphi) U_1 \\ (c_\theta c_\varphi) U_1 \\ U_2 \\ U_3 \\ U_4 \end{bmatrix} \quad \text{where} \quad \begin{aligned} U_1 &= F_1 + F_2 + F_3 + F_4 \\ U_2 &= l \cdot (-F_4 + F_2) \\ U_3 &= l \cdot (-F_1 + F_3) \\ U_4 &= l \cdot (F_1 - F_2 + F_3 - F_4) \end{aligned}$$

Equation 1.8: Input values for quadrocopter model

Out of Equation 1.8 it can be seen that only U_1 is a Force whereas e.g. U_2 is representing a torque around the roll axis (x-axis). See also chapter 1.1 where the different degrees of freedom are described.

As a second contribution to the input values the gravity force has to be respected, see Equation 1.9. Only the third component of this vector has a value because gravity is only affecting the z-direction.

$$G_h(\xi) = \begin{bmatrix} F_G^e \\ 0_{3 \times 3} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ -m g \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

Equation 1.9: Gravitational force

Overall speed differences of the rotors also have an influence to the quadrocopter because of gyroscopic effects. If there is an overall imbalance of the rotor speeds (the quadrocopter uses this principle for yawing) and furthermore an angular velocity in pitch/roll is present, this will result in a torque around the roll/pitch axis as it can be seen in Equation 1.10.

$$O_h(v) = \begin{bmatrix} 0_{3 \times 1} \\ J_{rotz} \begin{bmatrix} -q \\ p \\ 0 \end{bmatrix} \Omega \end{bmatrix} \quad \text{where} \quad \Omega = \Omega_1 - \Omega_2 + \Omega_3 - \Omega_4$$

Equation 1.10: Gyroscopic effects

Combining all these different inputs, gives the total input vector

$$\Lambda_h = U_h + G_h(\xi) + O_h(v)$$

Equation 1.11: Total input vector

which results in the equations of motion with respect to the hybrid frame.

$$\begin{aligned} \ddot{X} &= (s_\psi s_\varphi + c_\psi s_\theta c_\varphi) \frac{U_1}{m} \\ \ddot{Y} &= (-c_\psi s_\varphi + s_\psi s_\theta c_\varphi) \frac{U_1}{m} \\ \ddot{Z} &= -g + (c_\theta c_\varphi) \frac{U_1}{m} \\ \dot{p} &= \frac{I_y - I_z}{I_x} q r - \frac{J_{rotz}}{I_x} q \Omega + \frac{U_2}{I_x} \\ \dot{q} &= \frac{I_z - I_x}{I_y} p r + \frac{J_{rotz}}{I_y} p \Omega + \frac{U_3}{I_y} \\ \dot{r} &= \frac{I_x - I_y}{I_z} p q + \frac{U_4}{I_z} \end{aligned}$$

Equation 1.12: Equations of motion ($c_\psi = \cos\psi$, $s_\psi = \sin\psi$)

1.1.1 Implementation

For testing the control of the quadrocopter, the equations of motion, which have been derived in the previous section, were implemented with MATLAB/Simulink®. As seen in Figure 1.7 the block diagram can be split up into three major parts. The red part is responsible for processing the input data. In the blue area the equations of motion are calculated and the green area is in control of signal conditioning.

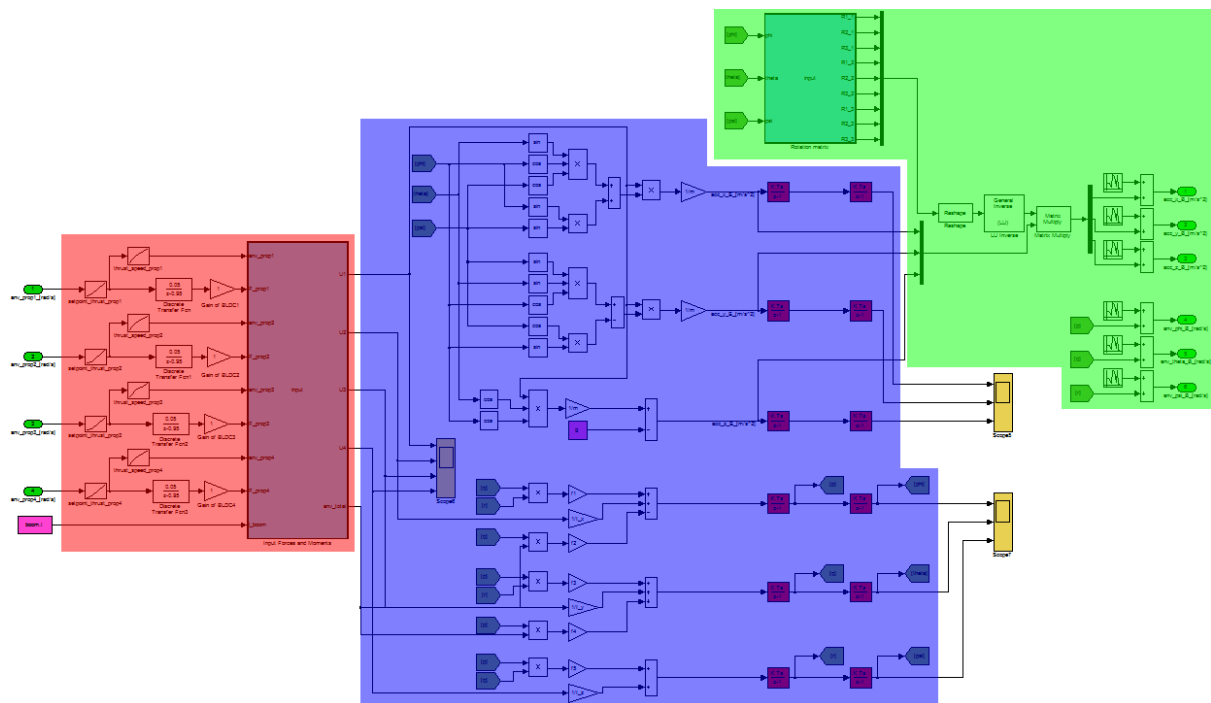


Figure 1.7: Implementation of the quadrocopter model

- Processing of input data

The quadcopter model gets four values as input. Each of them is to be seen as a value standing for a desired set point for the corresponding motor. At first the desired set point is transformed into an equivalent force by using a lookup table (see chapter 1.2 for more information about this curve), shown in Figure 1.8. Then the line splits in two branches. In the upper branch another lookup table transforms the force into a correlating speed for calculating the overall speed, see Equation 1.10. The lower branch consists of a transfer function and a gain block. These blocks are used to make the simulated model more realistic. The transfer function which implements a low pass behavior takes into account that the speed/force of the rotors cannot be changed at once. The following gain can be used to simulate a motor which does not produce the thrust is should with the desired set point. At last the inputs U_1 to U_4 for the equations of motion are calculated with an embedded MATLAB function using Equation 1.8.

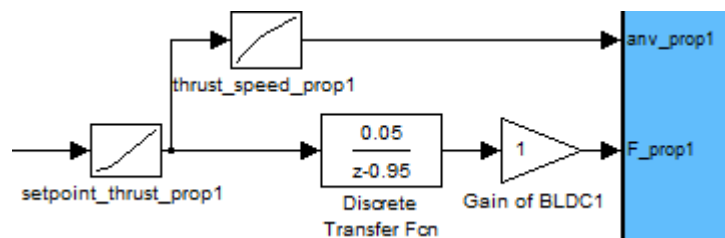


Figure 1.8: Processing of model input data

- The equations of motion

In the blue area the block diagram for the equations of motion (see Equation 1.12) is shown. The upper part is in charge of the linear movement, the lower half for determination of the angular motion. From top to bottom at first the acceleration in X_e -, Y_e - and Z_e -direction is calculated and afterwards integrated. The integration blocks are only for checking the results of the calculations; they are not needed in other parts of the model. In the lower half the calculation of the angular motion is implemented. By using from/goto blocks the wiring is reduced to keep a clear arrangement.

- Signal conditioning

In the real system there is no possibility to measure the accelerations with respect to the earth-fixed coordinate system because of the mounting point of the accelerometers. Therefore these signals have to be transformed to the body-fixed coordinate system. Using the actual angles during simulation, a rotation matrix (as it is described in Equation 1.4) is used. With an embedded MATLAB function the individual components of the rotation matrix are computed and then brought into a matrix representation using a reshape block. After that, a matrix inversion is calculated and the rotation matrix is finally multiplied with the acceleration signals. For further enhancement of the accuracy of the model white noise is added to the output of the model. This represents the actual behavior of real sensors.

1.2.3 Determination of the moment of inertia

The moment of inertia is important for simulating the dynamic behavior of the quadcopter. It describes the resistance of a rigid body against a change in the actual direction of rotation. Normally it is defined with a 3x3 matrix, also called tensor, but due to the assumption that the axes of the body-fixed coordinate system and the axes of the body inertia of the quadcopter are the same, only the diagonal entries (I_{xx} , I_{yy} , I_{zz}) in the inertia matrix have to be identified.

$$I = \begin{bmatrix} I_{xx} & I_{xy} & I_{xz} \\ I_{yx} & I_{yy} & I_{yz} \\ I_{zx} & I_{zy} & I_{zz} \end{bmatrix} = \begin{bmatrix} I_{xx} & 0 & 0 \\ 0 & I_{yy} & 0 \\ 0 & 0 & I_{zz} \end{bmatrix}$$

Equation 1.13: Diagonal inertia matrix

For determination of the inertia either a measurement can be performed by using a rotary table or it can be calculated by substituting the real parts of the quadcopter with simplified parts like cylinders or blocks. In this project the second approach was used.

1.2.4 Basic elements

- Cuboid

The measures of a cuboid are given by the edge lengths l (length), h (height) and w (width). For a rotation around the displayed axes in Figure 1.9 the inertia around the center of gravity is calculated by

$$I_x = \frac{1}{12} m \cdot (h^2 + d^2)$$
$$I_y = \frac{1}{12} m \cdot (w^2 + h^2)$$
$$I_z = \frac{1}{12} m \cdot (h^2 + d^2)$$

Equation 1.14: Inertia of cuboid

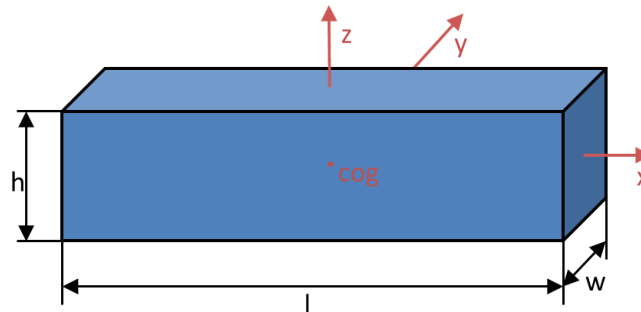


Figure 1.9: Inertia of cuboid

- Hemisphere

A hemisphere is given by its radius r , see Figure 1.10. The center of gravity is located $\frac{3}{8}r$ above the flattened part of the hemisphere. The inertia around the z-axis and around the other axes parallel to the flat part of the hemisphere are computed as shown in Equation 1.15.

$$I_z = \frac{2}{5}m \cdot r^2$$
$$I_x = I_y = \frac{83}{320}m \cdot r^2$$

Equation 1.15: Inertia of hemisphere

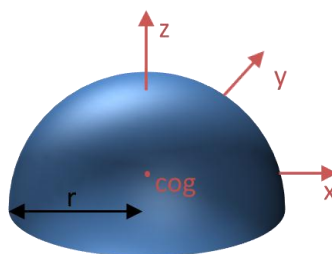


Figure 1.10: Inertia of hemisphere

- Cylinder

A cylinder is given by a radius and a height. The center of gravity is located at half of the height and in the middle of the circle. Its inertias around the principal axes are calculated by Equation 1.16.

$$I_z = \frac{1}{2} m \cdot r^2$$

$$I_x = I_y = \frac{1}{12} m \cdot (3r^2 + h^2)$$

Equation 1.16: Inertia of cylinder

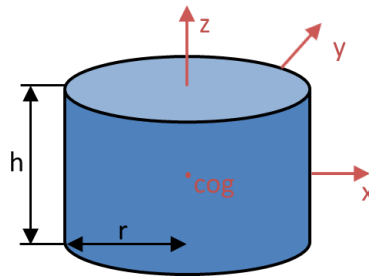


Figure 1.11: Inertia of cylinder

1.2.5 Steiner's parallel-axis theorem

Using the formulas shown in this section only the inertia around the center of gravity of each body is determined. In the abstracted model of the quadrocopter they take effect to the main center of gravity. Therefore Steiner's parallel-axis theorem has to be applied. It says that the inertia around the new axis is given by

$$I = I_{cm} + m \cdot d^2$$

Equation 1.17: Steiner's parallel-axis theorem

I_{cm} is the moment of inertia around the bodies center of gravity, m is the bodies mass and r is the perpendicular distance between the axis of the old and new center of gravity.

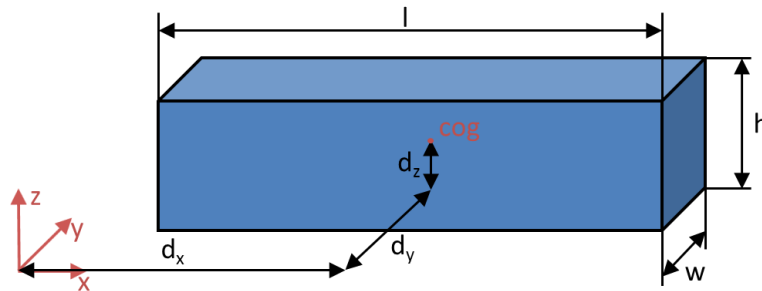


Figure 1.12: Steiner's parallel-axis theorem

1.2.6 Inertia of the simplified quadcopter model

For determination of the moment of inertia, a simplified model with the bodies of the previous section was built as shown in Figure 1.13. The structure consists of

- one hemisphere, which substitutes all the electronic parts (flight control, brushless controller, parts of the wiring harness and the construction plates)
- four motors, booms and propellers each substituted by cylinders
- one battery pack and the landing skids represented by a cuboid.

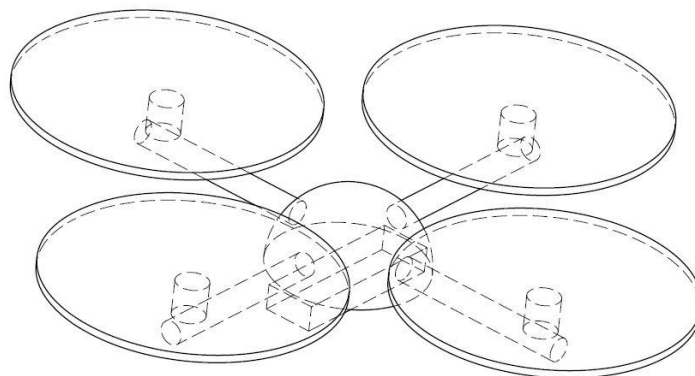


Figure 1.13: Simplified model of quadcopter

The hemispheric shaped electronics box has a radius of $63 \cdot 10^{-3} \text{m}$ and a mass of 0.25kg . In x_b - and y_b - direction it is located directly above the center of gravity, in z_b -direction there is a distance of $19.125 \cdot 10^{-3} \text{m}$. Figure 1.14 shows the geometry of the electronics box and Equation 1.18 the correspondent moments of inertia.

$$I_{elec_x} = \frac{83}{320} m_{elec} r_{elec}^2 + m_{elec} d_{elec_z}^2 = 3.4881 e^{-4} \text{ [Nms}^2\text{]}$$

$$I_{elec_y} = \frac{83}{320} m_{elec} r_{elec}^2 + m_{elec} d_{elec_z}^2 = 3.4881 e^{-4} \text{ [Nms}^2\text{]}$$

$$I_{elec_z} = \frac{2}{5} m_{elec} r_{elec}^2 = 3.969 e^{-4} \text{ [Nms}^2\text{]}$$

Equation 1.18: Inertia of the electronics box

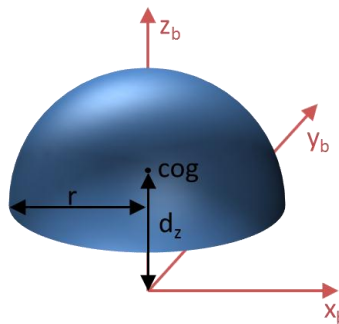


Figure 1.14: Inertia of the electronics box

As mentioned in the beginning of this section, the battery and the landing skids are modeled with a cuboid which has a length of 0.1435m , the width of $4.55 e^{-2} \text{m}$, a height of $2.3 e^{-2} \text{m}$ and a mass of 0.3kg . The center of gravity of the electronics box is shifted in z_b -direction relative to the origin of the body-fixed coordinate system by $2.35 e^{-2} \text{m}$.

The values for the moments of inertia are given in Equation 1.19. The geometry is shown in Figure 1.15.

$$I_{batx} = \frac{1}{12} m_{bat} (h_{bat}^2 + l_{bat}^2) + m_{bat} d_{batz}^2 = 6.9371e^{-4} \text{ [Nms}^2\text{]}$$

$$I_{baty} = \frac{1}{12} m_{bat} (h_{bat}^2 + w_{bat}^2) + m_{bat} d_{batz}^2 = 2.3066e^{-4} \text{ [Nms}^2\text{]}$$

$$I_{batz} = \frac{1}{12} m_{bat} (w_{bat}^2 + l_{bat}^2) = 5.6656e^{-4} \text{ [Nms}^2\text{]}$$

Equation 1.19: Inertia of the battery pack

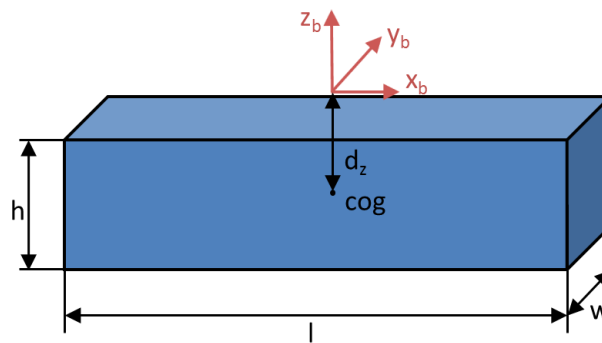


Figure 1.15: Inertia of the battery pack

The four booms show a high degree of symmetry. This symmetry can be used to make the calculations easier because only the inertia of one has to be computed and can be reused afterwards for the other booms. Each of the booms is replaced by a cylinder with a length of 0.166m, a radius of 1.15e⁻²m and a weight of 9e⁻²kg. For using Steiner's parallel axis theorem the perpendicular distance to the center of gravity of the copter is for boom 1 (boom on x-axis) 0.14m in x_b-direction and 1.7e⁻²m in z_b-direction.

Following values are representing a boom of the quadrocopter.

$$I_{boom1x} = \frac{1}{2} m_{boom} r_{boom}^2 + m_{boom} d_{boomz}^2 = 3.1961e^{-5} [Nms^2]$$

$$I_{boom1y} = \frac{1}{16} m_{boom} \left(4 r_{boom}^2 + \frac{4}{3} l_{boom}^2 \right) + m_{boom} (d_{boomx}^2 + d_{boomz}^2) = 2e^{-3} [Nms^2]$$

$$I_{boom1z} = \frac{1}{16} m_{boom} \left(4 r_{boom}^2 + \frac{4}{3} l_{boom}^2 \right) + m_{boom} d_{boomx}^2 = 2e^{-3} [Nms^2]$$

Equation 1.20: Inertia of boom one

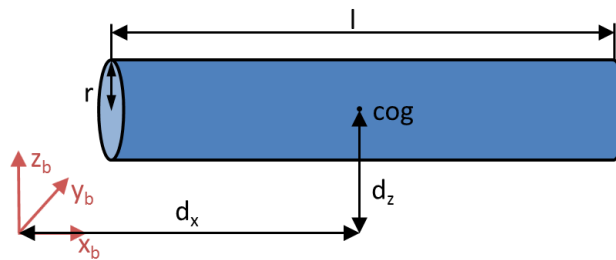


Figure 1.16: Inertia of boom one

Due to symmetry in the construction the values for the other booms are given by

$$I_{boom1x} = I_{boom2y} = I_{boom3x} = I_{boom4y}$$

$$I_{boom1y} = I_{boom2x} = I_{boom3y} = I_{boom4x}$$

$$I_{boom1z} = I_{boom2z} = I_{boom3z} = I_{boom4z}$$

Equation 1.21: Symmetry conditions for booms

The motors are also represented by cylinders. As it was the case with the booms, also the motors show a high degree of symmetry. The basic measures of each cylinder are a height of $3.15e^{-2}m$, a radius of $1.42e^{-2}m$ and a mass of $0.1kg$. The distance to the center of gravity for motor 1 is in x_b -direction $0.2025m$ and in z_b -direction $4.2e^{-2}m$.

Figure 1.17 shows the geometry of one motor, the values for the resulting inertias are given in Equation 1.22.

$$I_{mot1x} = \frac{1}{16} m_{mot} \left(4 r_{mot}^2 + \frac{4}{3} h_{mot}^2 \right) + m_{mot} d_{motz}^2 = 1.897e^{-4} \text{ [Nms}^2\text{]}$$

$$I_{mot1y} = \frac{1}{16} m_{mot} \left(4 r_{mot}^2 + \frac{4}{3} h_{mot}^2 \right) + m_{mot} (d_{motx}^2 + d_{motz}^2) = 4.3e^{-3} \text{ [Nms}^2\text{]}$$

$$I_{mot1z} = \frac{1}{2} m_{mot} r_{mot}^2 + m_{mot} d_{motx}^2 = 4.1e^{-3} \text{ [Nms}^2\text{]}$$

Equation 1.22: Inertia of motor one

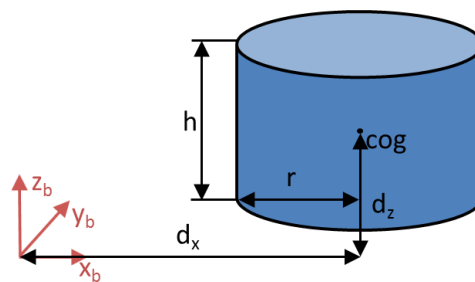


Figure 1.17: Inertia of motor one

Thanks to symmetry of the structure the inertias of the other motors are

$$I_{mot1x} = I_{mot2y} = I_{mot3x} = I_{mot4y}$$

$$I_{mot1y} = I_{mot2x} = I_{mot3y} = I_{mot4x}$$

$$I_{mot1z} = I_{mot2z} = I_{mot3z} = I_{mot4z}$$

Equation 1.23: Symmetry conditions for motors

The last objects which have to be included are the rotors. Each rotor is represented by a cylinder which has a height of $1e^{-2}m$, a radius of $0.127m$ and a weight of $1e^{-2}kg$. The perpendicular distances of propeller 1 to the center of mass are in x_b -direction $0.2025m$ and in z_b -direction $6.3e^{-2}m$.

Figure 1.18 shows the geometry of one rotor, the values for the calculated inertias are the following.

$$I_{rot1x} = \frac{1}{16} m_{rot} \left(4 r_{rot}^2 + \frac{4}{3} h_{rot}^2 \right) + m_{rot} d_{rotz}^2 = 8.0096e^{-5} \text{ [Nms}^2\text{]}$$

$$I_{rot1y} = \frac{1}{16} m_{rot} \left(4 r_{rot}^2 + \frac{4}{3} h_{rot}^2 \right) + m_{rot} (d_{rotx}^2 + d_{rotz}^2) = 4.9016e^{-4} \text{ [Nms}^2\text{]}$$

$$I_{rot1z} = \frac{1}{2} m_{rot} r_{rot}^2 + m_{rot} d_{rotx}^2 = 4.9071e^{-4} \text{ [Nms}^2\text{]}$$

Equation 1.24: Inertia of rotor one

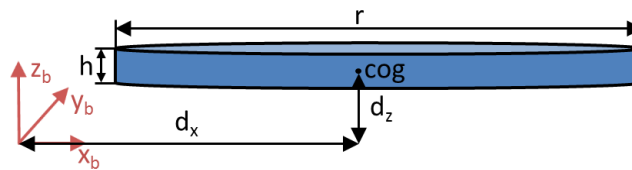


Figure 1.18: Inertia of rotor one

Due to symmetry there is a correlation between each rotor.

$$I_{rot1x} = I_{rot2y} = I_{rot3x} = I_{rot4y}$$

$$I_{rot1y} = I_{rot2x} = I_{rot3y} = I_{rot4x}$$

$$I_{rot1z} = I_{rot2z} = I_{rot3z} = I_{rot4z}$$

Equation 1.25: Symmetry conditions for rotors

For the equations of motion not only the inertias with respect to the complete structure are needed but also the inertia around its own center of mass, see Equation 1.10. In this case the rotor is not represented as a disc anymore but as a cuboid with the measures of height equal to $1e^{-2}\text{m}$, a width of $1e^{-2}\text{m}$ and a length of 0.254m .

This results in inertia around z-axis of

$$J_{rotz} = \frac{1}{12} m_{rot} (w_{rot}^2 + l_{rot}^2) = 5.3847e^{-5} \text{ [Nms}^2\text{]}$$

Equation 1.26: Inertia of rotor around rotors center of mass

After calculating each simplification of the structure, the evaluation of the complete inertia can be done by adding all components around its specific axis as shown in Equation 1.27.

$$I_x = I_{elec} + I_{bat} + I_{boom1x} + I_{boom2x} + I_{boom3x} + I_{boom4x} + I_{mot1x} + I_{mot2x} + I_{mot3x} + I_{mot4x} + I_{rot1x} + I_{rot2x} + I_{rot3x} + I_{rot4x} = 1.52e^{-2} \text{ [Nms}^2\text{]}$$

$$I_y = I_{elec} + I_{bat} + I_{boom1y} + I_{boom2y} + I_{boom3y} + I_{boom4y} + I_{mot1y} + I_{mot2y} + I_{mot3y} + I_{mot4y} + I_{rot1y} + I_{rot2y} + I_{rot3y} + I_{rot4y} = 1.47e^{-2} \text{ [Nms}^2\text{]}$$

$$I_z = I_{elec} + I_{bat} + I_{boom1z} + I_{boom2z} + I_{boom3z} + I_{boom4z} + I_{mot1z} + I_{mot2z} + I_{mot3z} + I_{mot4z} + I_{rot1z} + I_{rot2z} + I_{rot3z} + I_{rot4z} = 9.937e^{-3} \text{ [Nms}^2\text{]}$$

Equation 1.27: Complete inertia of quadcopter

The values of I_x and I_y are almost the same which indicates the symmetry of the quadcopter. The small differences only come from the asymmetric battery pack. Compared to the values of I_x and I_y , I_z is relatively small. This means that the quadcopters resistance against a yawing is less than the resistance against pitching and rolling.

1.2 Determination of the thrust-set point curve

The controller of the quadcopter has a desired force as output. These forces cannot be used as set points for the brushless controller because each quadcopter can have a different setup with different motors (which will produce diverse speeds at a specific current) and different rotors (which will generate a different thrust for a specific speed). Therefore there is a need to create a curve which represents the transformation of a desired force into a set point which is then sent to the brushless controllers. In order to achieve this correlation between force and set point, a test rig was used. In this test rig every DOF except for one is blocked. Only movement around one axis, pitch or roll, is possible and only one motor is running. In the experiment a mass of known weight was attached to this motor. Now the motor was accelerated by the remote control until a horizontal plane was realized. Then the actual set point, the compensated weight force of the attached mass and the battery voltage was noted down. This experiment was repeated for different weights and different battery voltages. The results are shown in Figure 1.19.

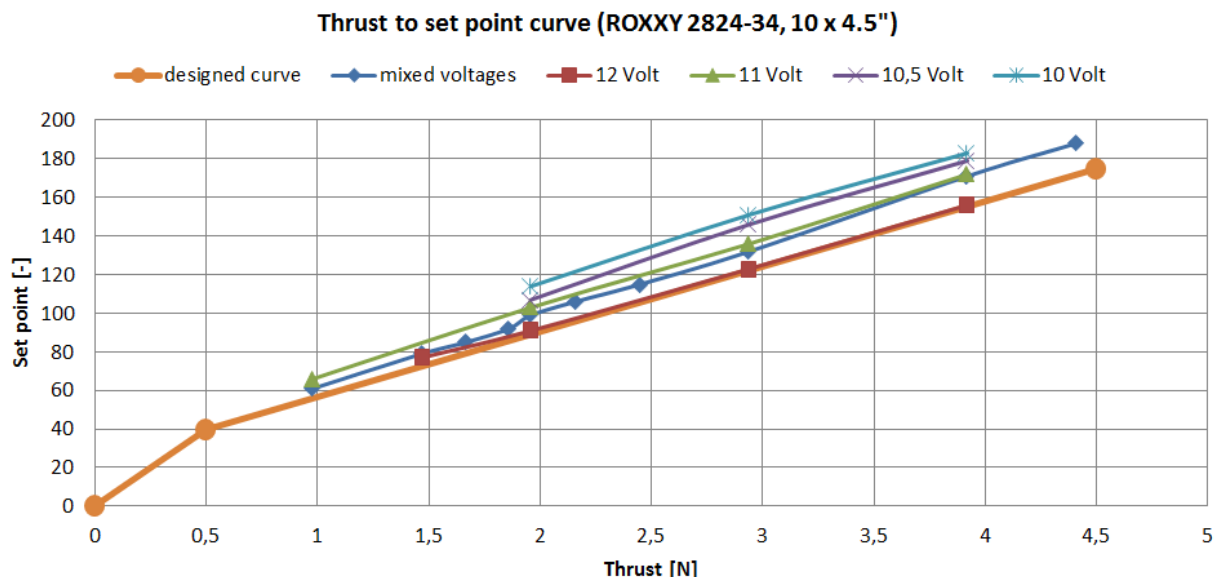


Figure 1.19: Thrust to set point curve

The figure above shows that there is a dependency of the produced thrust with the battery voltage but it also shows that there is a linear behavior. With these two observations a function, to derive the needed set point from a desired force, can be developed. As a basis, a curve for a fully charged battery is created using and interpolating the 12V curve from Figure 1.19. This designed curve is displayed as orange line in the same figure. For the dependency of the battery voltage a percentaged deviation is assumed. This means a 20% gain of the set point for a battery voltage of 10V from the original 12V set point.

An additional curve describes the correlation between the forces of a rotor with a specific speed. This curve was measured by the students from Göppingen. See Figure 1.20 for the relationship.

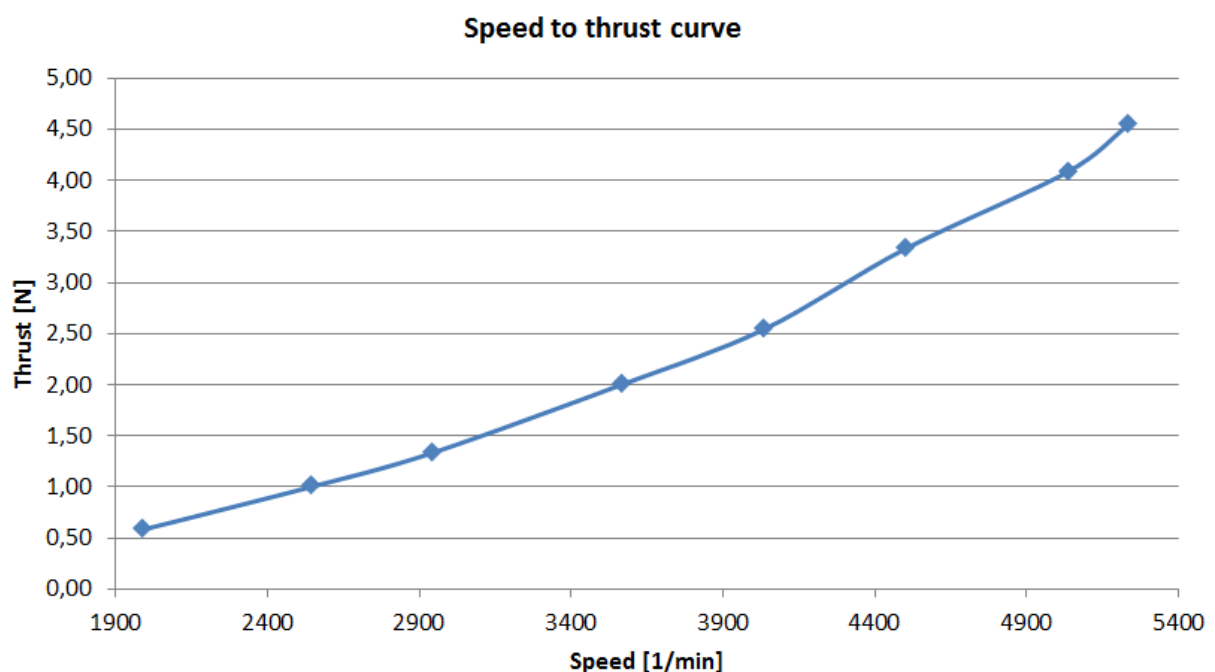


Figure 1.20: Speed to thrust curve

1.3 Improvements and conclusion

The model itself was tested with test cases, giving for example a constant input speed for all rotors and observing the result. Also more complicated maneuvers, which include pitch, roll and yaw movement, were tested. For checking the outputs of the model, a m-file which visualizes the movement and the attitude of the simulated quadcopter model was created. These results looked satisfactory but could not be verified with the existing real quadcopter.

A possible way to verify the behavior of the model could be to test the model with real input values and compare the outputs (sensor signals) of the model with those of the real and flying quadcopter. This can be accomplished using the same signals of the remote. By connecting the remote control with an adapter to a PC and running a program (freely available on the internet), the inputs of the remote can be interpreted as joystick information. These values then can be used for the MATLAB/Simulink® simulation. After simulation the recorded signals and the transmitted signals from the quadcopter can be compared. This allows verifying the behavior of the model.

2 Quadcopter Control Algorithm and Sensor Filtering

2.1 Control Algorithm

In this chapter, the basic control of the quadcopter (Flight control) is presented. In the first step, the control algorithm is developed by MATLAB/Simulink® basic blocks & tested with some probabilistic & “near-real-life/real-time” test cases. The control algorithm (“Controller”) is interfaced with the quadcopter model (“Plant”) (which is also developed by MATLAB/Simulink® basic blocks). Together with this arrangement, a filtering algorithm (“Sensor Filter”) is later developed in the feedback loop to filter the sensors values for its “drifts”.

For the simulations, the optimized values for control parameters were “auto-tuned” with the MATLAB/Simulink’s® in-built tuning functions.

Later in the second step, tests were carried out on the quadcopter platform to evaluate the behavior of the real system.

As this chapter is closely coupled with the chapter “Quadcopter System Model”, we try to analyze the quadcopter model itself & attempt to “invert” some of the equations to control the quadcopter.

The first section (Control Modelling) shows the basic quadcopter model simplifications. These must be done to be able to use an easier controller and to lower the algorithm complexity.

The second section (PID Techniques) introduces the PID concepts and its strengths. After that it shows and explains in detail the 3 inner control diagrams. Their goal is to determine the basic movement signals from accelerometer and gyros data (sensors) and from task references (remote controller).

2.2 Control Modelling

Since the quadcopter model is described in the previous chapter in sufficient detail, here we summarize most important equations.

The first equation shows, how the quadcopter accelerates according to the basic movement commands given.

$$\begin{aligned}\ddot{X} &= (\sin \Psi \sin \Phi + \cos \Psi \sin \theta \cos \Phi) \frac{U_1}{m} \\ \ddot{Y} &= (-\cos \Psi \sin \Phi + \sin \Psi \sin \theta \cos \Phi) \frac{U_1}{m} \\ \ddot{Z} &= -g + (\cos \theta \cos \Phi) \frac{U_1}{m} \\ \dot{p} &= \frac{I_{YY} - I_{ZZ}}{I_{XX}} qr - \left(\frac{J_{TP}}{I_{XX}} \right) q\Omega + \frac{U_2}{I_{XX}} \\ \dot{q} &= \frac{I_{ZZ} - I_{XX}}{I_{YY}} pr + \left(\frac{J_{TP}}{I_{YY}} \right) p\Omega + \frac{U_3}{I_{YY}} \\ \dot{r} &= \frac{I_{XX} - I_{YY}}{I_{ZZ}} pq + \frac{U_4}{I_{ZZ}}\end{aligned}$$

Equation 2.1: Equations of Motion

The second system of equations explains how the basic movements are related to the forces to the individual propellers.

$$\begin{aligned}U_1 &= F_1 + F_2 + F_3 + F_4 \\ U_2 &= l(F_2 - F_4) \\ U_3 &= l(F_3 - F_1) \\ U_4 &= F_1 - F_2 + F_3 - F_4\end{aligned}$$

Equation 2.2: Basic movements versus individual propellers forces

The propellers' forces are transformed into the speed of the individual rotors using a transformation curve characteristics (from the manufacturer), since the BLDC motors expect the rotor speed as an input to control the speed of the individual rotors.

From these set of equations, it is possible to determine the quadrocopter position by integrating the accelerations (linear & angular) twice. To do this operation, just the internal state of the quadrocopter and the speed of the four motors must be managed. This process is also known as direct kinematics and direct dynamics.

Controlling of the quadrocopter is done by finding the right values of the speed of the motors, which maintains the quadrocopter in a stable position commanded by the task references. To do this, one should invert the model characteristics & equations. This process is also known as inverse kinematics and inverse dynamics. Since the inverse operations are bit complicated to perform, we present here some simplified inverse equations from (Equation 2.1: Equations of Motion)

Since we can only command the Roll, Pitch & Yaw angles from the remote controller, we try to control the angular accelerations of Roll, Pitch & Yaw values. For these, 3 out of 6 equations described in the (Equation 2.1: Equations of Motion) is sufficient to simplify & invert. Hence we get the following set of equations.

$$\begin{aligned}\ddot{\Phi} &= \frac{U_2}{I_{XX}} \\ \ddot{\Theta} &= \frac{U_3}{I_{YY}} \\ \ddot{\Psi} &= \frac{U_4}{I_{ZZ}}\end{aligned}$$

Equation 2.3: Simplified & Inverted Equations of motion

Or

$$U_2 = \ddot{\phi} I_{XX}$$

$$U_3 = \ddot{\theta} I_{YY}$$

$$U_4 = \ddot{\psi} I_{ZZ}$$

Equation 2.4: Alternate form of Simplified & Inverted Equations of motion

The control algorithm receives, as inputs, the data from the sensors and from the remote controller. The output of the control algorithm is the speed for the four motors.

The control algorithm itself contains 2 blocks – “Inner Control Algorithm” & “Inverted Movement Matrix”

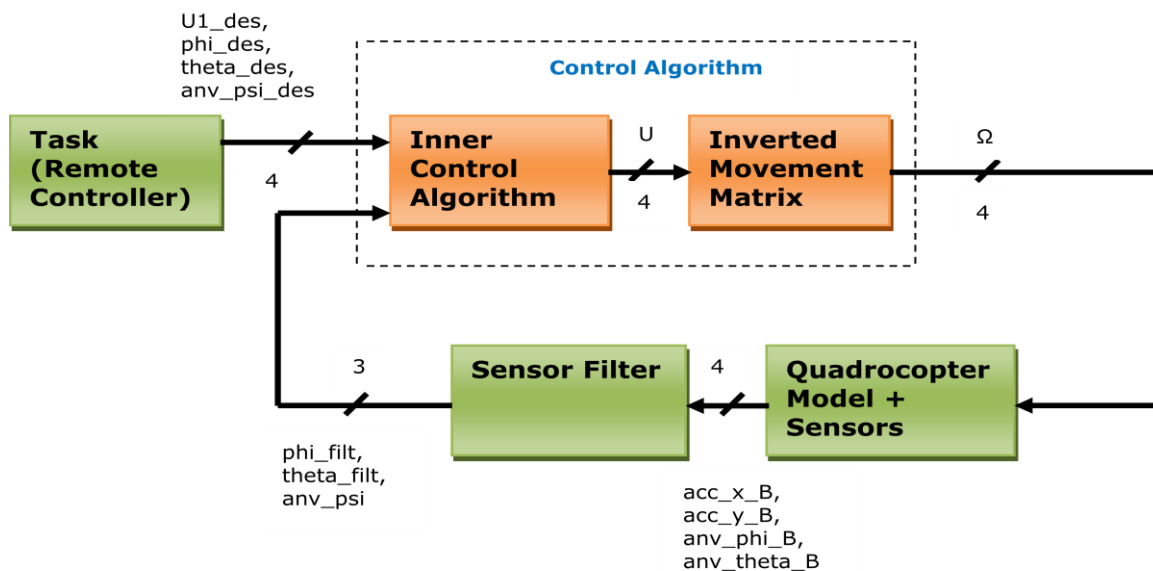


Figure 2.1: Control Block Diagram

- “Inner Control Algorithm” represents the core PID controllers. It processes the command received from the *remote controller* & *sensor data* to form the error signal to each of the PID controller. Output of each PID controller gives a signal for each of the basic movements which balances the position error
- “Inverted Movement Matrix” represents a block, which calculates the forces for the individual propellers from the 4 basic movement signals. This is obtained by inverting the (Equation 2.2: Basic movements versus individual propellers forces)

$$\begin{aligned}F_I &= \frac{U_I}{4} - \frac{U_3}{2l} + \frac{U_4}{4l} \\F_2 &= \frac{U_I}{4} + \frac{U_2}{2l} - \frac{U_4}{4l} \\F_3 &= \frac{U_I}{4} + \frac{U_3}{2l} + \frac{U_4}{4l} \\F_4 &= \frac{U_I}{4} - \frac{U_2}{2l} - \frac{U_4}{4l}\end{aligned}$$

Equation 2.5: Equations for forces of individual propellers

2.3 PID Techniques

In most of the general industrial applications, the most widely used controllers are the PID controllers because of the following reasons,

- Simple & robust structure
- Good performance for several processes
- Adjustable even without a specific model of the controlled system

Hence we have also chosen the PID controllers in our project to control the quadcopter.

So the quadcopter is constantly controlled by a fast digital feedback loop. The sensors are read and then the error is calculated.

Since the control loop is the PID controller, the output is controlled by 3 components as shown in the Figure 2.2: PID Structure.

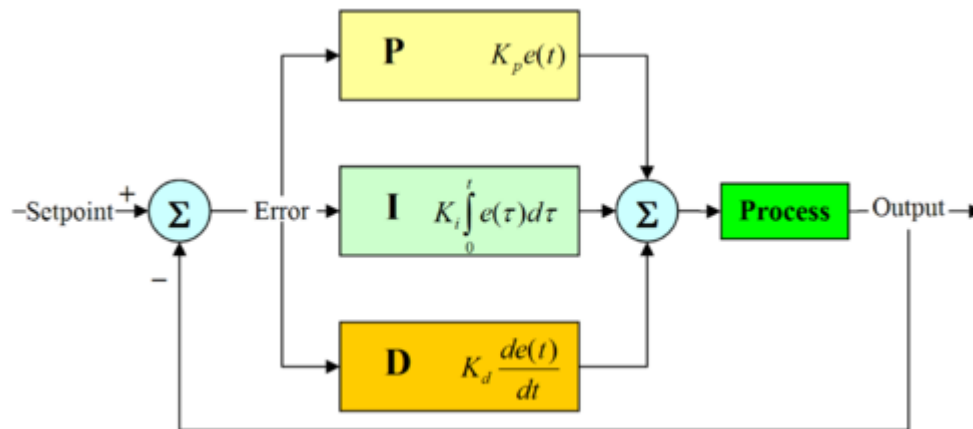


Figure 2.2: PID Structure

- Proportional: This is the error in output multiplied by a number. This defines the proportional bandwidth.
- Integral: This is the integral value of the output. In other words, by constantly adding all errors together it is possible to create a steady state error of zero, even though this component increases the overshoot & the settling time.
- Differential: This is the differential value of the output. In other words, taking the change in error, fast changes (wind gusts) in the error are corrected by this value. This component helps to decrease the overshoot & the settling time.

The equation governing the PID controller is as below:

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

Equation 2.6: Basic PID equation in Time domain

Where $u(t)$ is a generic controlled variable, $e(t)$ is the error between the "Setpoint" and the "Process output", K_p is the proportional coefficient, K_i is the integral coefficient and K_d is the derivative coefficient.

The description of the 3 inner control algorithms needed to control the Roll, Pitch & the Yaw are described in the following section

2.3.1 Roll Control

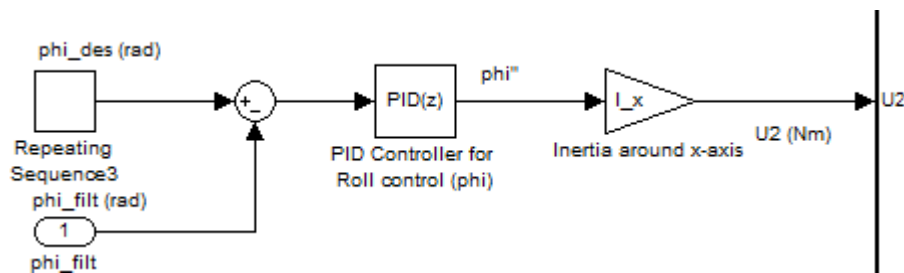


Figure 2.3: Block diagram of the Roll Control

$\phi_{des} [rad]$ represents the desired roll angle commanded from the remote controller; $\phi_{filt} [rad]$ represents the filtered roll angle derived out of the "Sensor Filter" block. The difference between these 2 signals gives the error signal for the roll angle which is then given to the PID controller. The output of the controller is a roll angular acceleration ($\ddot{\phi}$) which is then multiplied by the body moment of inertia around the x-axis $I_{xx} [Nms^2]$ to get the required roll torque $U_2[Nm]$. This inertia comes from the (Equation 2.4: Alternate form of Simplified & Inverted Equations of motion) & is needed to relate the roll control to U_2 .

$Kp_{\phi} [1/s^2]$, $Ki_{\phi} [1/s^3]$ & $Kd_{\phi} [1/s]$ represents the control parameters for the roll control.

2.3.2 Pitch Control

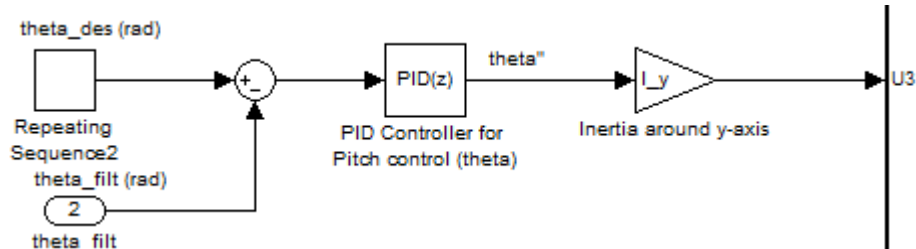


Figure 2.4: Block diagram of the Pitch Control

$\theta_{des} [rad]$ represents the desired pitch angle commanded from the remote controller; $\theta_{filt} [rad]$ represents the filtered pitch angle derived out of the "Sensor Filter" block. The difference between these 2 signals gives the error signal for the pitch angle which is then given to the PID controller. The output of the controller is a pitch angular acceleration ($\ddot{\theta}$) which is then multiplied by the body moment of inertia around the y-axis $I_{yy} [Nms^2]$ to get the required pitch torque $U_3 [Nm]$. This inertia comes from the (Equation 2.4: Alternate form of Simplified & Inverted Equations of motion) & is needed to relate the pitch control to U_3 .

$Kp_{\theta} [1/s^2]$, $Ki_{\theta} [1/s^3]$ & $Kd_{\theta} [1/s]$ represents the control parameters for the pitch control.

2.3.3 Yaw Rate Control

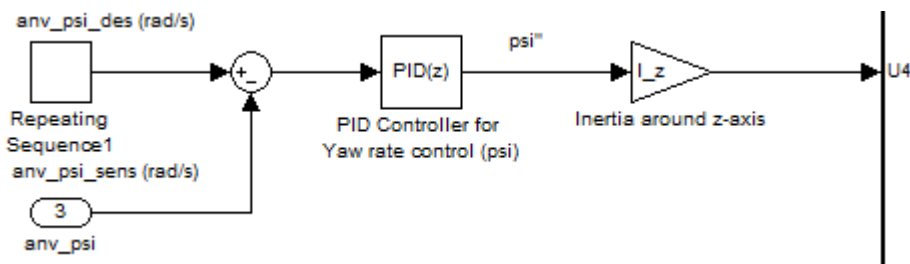


Figure 2.5: Block diagram of the Yaw Rate Control

anv_psi_des [rad/s] represents the desired yaw rate commanded from the remote controller; anv_psi_sens [rad/s] represents the actual yaw rate measured by the gyroscope. The difference between these 2 signals gives the error signal for the yaw rate which is then given to the PID controller. The output of the controller is a yaw angular acceleration ($\ddot{\psi}$) which is then multiplied by the body moment of inertia around the z-axis I_{zz} [Nms²] to get the required yaw torque U_4 [Nm]. This inertia comes from the (Equation 2.4: Alternate form of Simplified & Inverted Equations of motion) & is needed to relate the yaw rate control to U_4 .

Kp_anv_psi [1/s], Ki_anv_psi [1/s²] & Kd_anv_psi [-] represents the control parameters for the yaw rate control.

2.4 Sensor Filtering

Before explaining what "Sensor Filtering" means, let us look into sensors we use & what values we get from these sensors. Currently in our project we use 1 accelerometer (LIS3LV02DQ) which is capable of measuring the acceleration in all the axes (X-, Y- & Z- axis) & 3 gyros (ADXRS610) which is capable of measuring the angular rates (angular velocities roll, pitch & yaw).

One accelerometer & three gyros, together form what is known as "Inertial Measuring Unit (IMU)". The purpose of the IMU is to constantly keep track of the position and angle of the quadrocopter.

But unfortunately the gyros suffer from what is known as "drifts" i.e. over long term these gyros will not provide correct angular rate signals. They are prone to noise & produce incorrect results over a period of time. To overcome this problem, acceleration signals from accelerometer is combined with the angular rate signals from the gyros. The gyros will provide very good results for the short term and the long-term drift of the gyros is eliminated by the accelerometer.

With this background, we are now in a position to define what "Sensor Filtering" means. It simply means combining 2 different measurement sources to "estimate" one variable by appropriately choosing "weighting factors" for the 2 different sources. This is realized in what is known as "Complementary Filter" or "Balance Filter".

2.4.1 Complementary Filter

Often, there are cases where we have two different measurement sources for estimating one variable and the noise properties of the two measurements are such that one source gives good information only in low frequency region while the other is good only in high frequency region.

A "Complementary Filter" is a simple method for integrating the accelerometer & gyros measurements for achieving a balanced platform. Hence it is sometimes also known as "Balanced Filter".

2.4.2 Calculation of the angle from x-axis accelerometer value

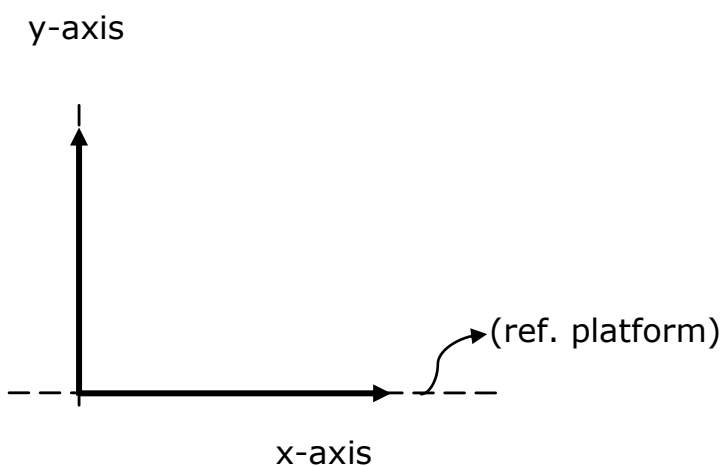


Figure 2.6: Quadcopter lying on the reference platform

If the x-axis of the quadcopter lies exactly on the reference platform, then the value read by the accelerometer (x-axis) = 0 g.

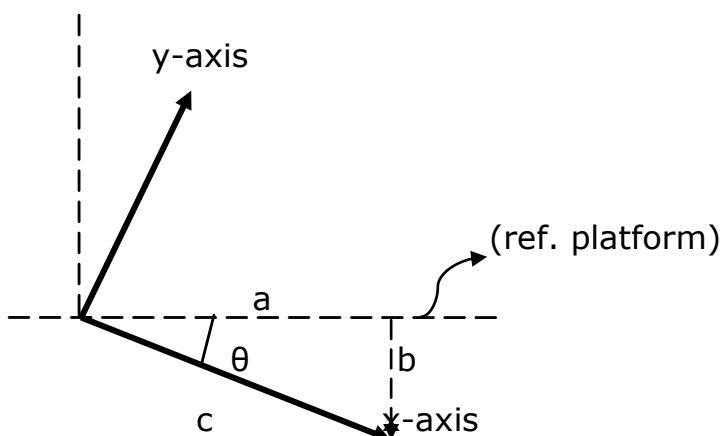


Figure 2.7: Quadcopter tilted w.r.t. the reference platform

$$\sin \theta = \frac{b}{c}$$

where

- c is $1g$
- b is a component of $1g$ (i.e. acceleration value around x-axis from the sensor)
- θ is the tilt angle in radians

For small angles we can approximate $\sin \theta = \theta$. Hence, $\theta = \frac{b}{1g}$.

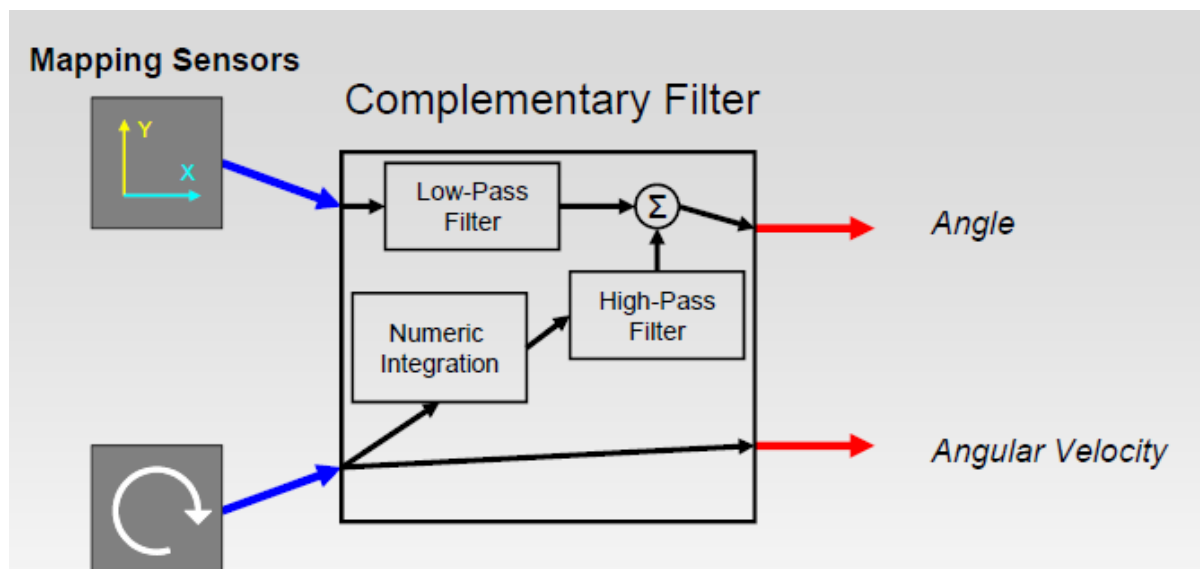


Figure 2.8: Fusion of Accelerometer & Gyro to get the filtered angle estimate

With this arrangement, it is now easier to calculate the angle estimate by this simple formula which is depicted in the figure shown above

$$angle = (HPF) * (angle + gyro * dt) + (LPF) * (x_{acc})$$

Equation 2.7: Complementary Filter Equation

The 3 essential parts of the Complementary Filter are the "Numeric Integration", "Low-Pass Filter" & the "High-Pass Filter".

- Numerical Integration: Gyros give the angular rates (or velocities). To get the angle information at each sample time dt , we take previous *angle* information & then add the change in angle to get the new *angle* information.

For the balancing platform this is given by

$$angle = (angle + gyro * dt)$$

Equation 2.8: Numerical integration part of the equation

- Low-Pass filter: This part of the Complementary Filter ensures that only the long-term changes are passed through it by filtering out the short term fluctuations.

$$angle = (LPF) * (x_{acc})$$

Equation 2.9: Low-Pass filter part of the equation

- High-Pass filter: This does exactly the opposite to the Low-Pass filter i.e. it allows short-duration signals to pass through while filtering out signals that are steady over time. This can be used to cancel out drift.

$$angle = (HPF) * (angle + gyro * dt)$$

Equation 2.10: High-Pass filter part of the equation

The above 3 (Equation 2.8), (Equation 2.9) & (Equation 2.10) are combined to get the complete equation for the Complementary filter.

Besides these 3 parts, 2 other relatively important parts influence the response from the Complementary filter – “Sample Time” & “Time Constant”.

- **Sample Time:** This is the amount of time between each successive program loop execution of the control algorithm. If the sample rate is 100Hz, the sample time is 10ms.
- **Time Constant:** The time constant of a filter is the relative duration of signal it will act on. For a low-pass filter, signals much longer than the time constant pass through unaltered while signals shorter than the time constant are filtered out. The opposite is true for a high-pass filter. The time constant, τ , of a digital low-pass filter, $y = (a) * (y) + (1-a) * (x)$, running in a loop with sample period, dt , can be found like this:

$$\tau = a \frac{dt}{1-a} \leftrightarrow a = \frac{\tau}{\tau + dt}$$

Equation 2.11: Time constant of the filter

2.5 Model Transformation into Embedded Version Realization

Until now we have described the mathematical equations & system blocks as shown in (Figure 2.1: Control Block Diagram) in terms of the Simulink® block diagrams. Since we do not intend to generate the code from the MATLAB automatically (since it would be little difficult to comprehend its syntaxes & semantics from the auto code!), we plan to manually hand-code the function blocks that were built in Simulink®. For that we need to transform these blocks into a suitable embedded version with the appropriate discretization applied to the signals, so that we get a proper integer representation of all the signals with reasonable resolutions (accuracies). This is realized from the MATLAB's S-Functions feature.

2.5.1 Brief Info about S-Functions

An S-function is a computer language description of a dynamic system. S-functions can be written using MATLAB or C. C language S-functions are compiled as MEX-files using the *mex* utility or as DLL files using *Visual Studio*. As with other MEX-files, they are dynamically linked into MATLAB when needed. S-functions use a special calling syntax that enables us to interact with Simulink®'s equation solvers. This interaction is very similar to the interaction that takes place between the solvers and built-in Simulink® blocks.

The form of an S-function is very general and can accommodate continuous, discrete, and hybrid systems. As a result, nearly all Simulink® models can be described as S-functions.

MATLAB S-functions are an effective way to embed object code into a Simulink® model. These S functions can be written in a few languages, C being the one most relevant for our purposes.

2.5.2 Simulation Stages and S-Function Routines

Simulink® makes repeated calls during specific stages of simulation to each block in the model, directing it to perform tasks such as computing its outputs, updating its discrete states, or computing its derivatives. Additional calls are made at the beginning and end of a simulation to perform initialization and termination tasks.

The figure below illustrates how Simulink® performs a simulation. First, Simulink® initializes the model; this includes initializing each block, including S-functions. Then Simulink® enters the simulation loop, where each pass through the loop is referred to as a simulation step. During each simulation step, Simulink® executes our S-function block. This continues until the simulation is complete:

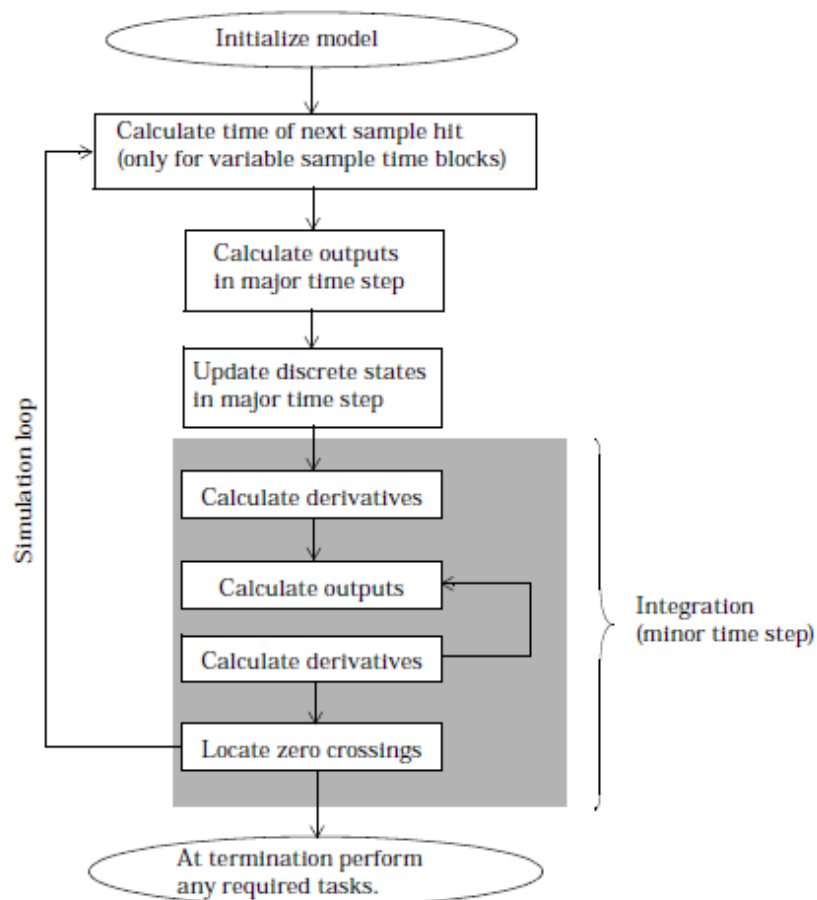


Figure 2.9: How Simulink® Performs Simulation

Simulink® makes repeated calls to S-functions in the model. During these calls, Simulink® calls S-function routines (also called methods), which perform tasks required at each stage. These tasks include:

- Initialization — Prior to the first simulation loop, Simulink® initializes the S-function. During this stage, Simulink®:
 - Initializes the *SimStruct*, a simulation structure that contains information about the S-function
 - Sets the number and size of input and output ports
 - Sets the block sample time(s)
 - Allocates storage areas and the sizes array
- Calculation of next sample hit — If you've selected a variable step integration routine, this stage calculates the time of the next variable hit, that is, it calculates the next step size
- Calculation of outputs in the major time step — After this call is complete, all the output ports of the blocks are valid for the current time step
- Update discrete states in the major time step — In this call, all blocks should perform once-per-time-step activities such as updating discrete states for next time around the simulation loop
- Integration — This applies to models with continuous states and/or non-sampled zero crossings. If the S-function has continuous states, Simulink® calls the output and derivative portions of the S-function at minor time steps. This is so Simulink® can compute the state(s) for the S-function. If the S-function (C MEX only) has non-sampled zero crossings, then Simulink® will call the output and zero crossings portion of the S-function at minor time steps, so that it can locate the zero crossings

2.6 Implemented S-Function blocks of Quadcopter

Following S-Function blocks are implemented in Simulink®'s S-Function (C-Code).

- Quadcopter Controller
- Sensor Filter

2.6.1 S-Function block Quadcopter Controller

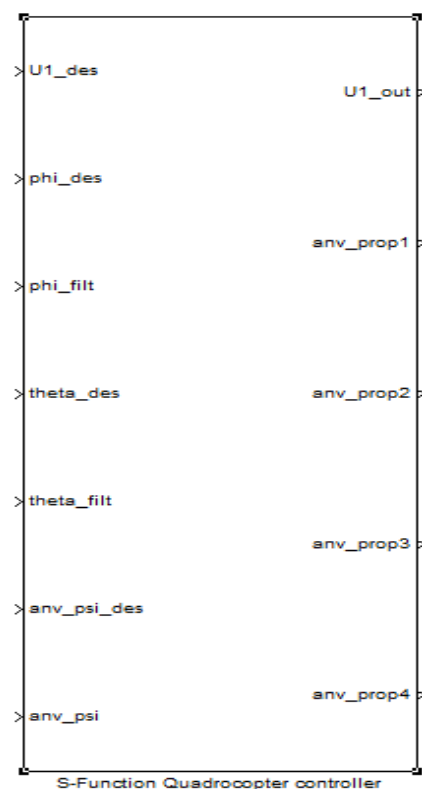


Figure 2.10: S-Function Quadcopter controller

- Inputs
 - U1_des: Desired total force commanded from the remote controller
 - phi_des: Desired roll angle commanded from the remote controller

- `phi_filt`: Filtered roll angle as indicated by the sensors (accelerometer & gyro)
 - `theta_des`: Desired pitch angle commanded from the remote controller
 - `theta_filt`: Filtered pitch angle as indicated by the sensors (accelerometer & gyro)
 - `anv_psi_des`: Desired yaw velocity commanded from the remote controller
 - `anv_psi`: yaw velocity as indicated by the sensor (gyro)
- Outputs
 - `U1_out`: Total force output from all the propellers
 - `anv_prop1`: Speed of propeller 1 (Front propeller)
 - `anv_prop2`: Speed of propeller 2 (Left propeller)
 - `anv_prop3`: Speed of propeller 3 (Rear propeller)
 - `anv_prop4`: Speed of propeller 4 (Right propeller)

2.6.2 S-Function block Sensor Filter

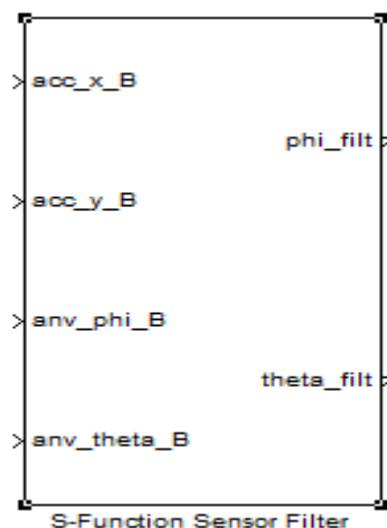


Figure 2.11: S-Function Sensor Filter

- Inputs
 - acc_x_B: Acceleration of the Body frame in the X direction as indicated by the sensor (accelerometer)
 - acc_y_B: Acceleration of the Body frame in the Y direction as indicated by the sensor (accelerometer)
 - anv_phi_B: Roll velocity as indicated by the sensor (gyro)
 - anv_theta_B: Pitch velocity as indicated by the sensor (gyro)

- Outputs:
 - phi_filt: Filtered roll angle as indicated by the sensors (accelerometer & gyro)
 - theta_filt: Filtered pitch angle as indicated by the sensors (accelerometer & gyro)

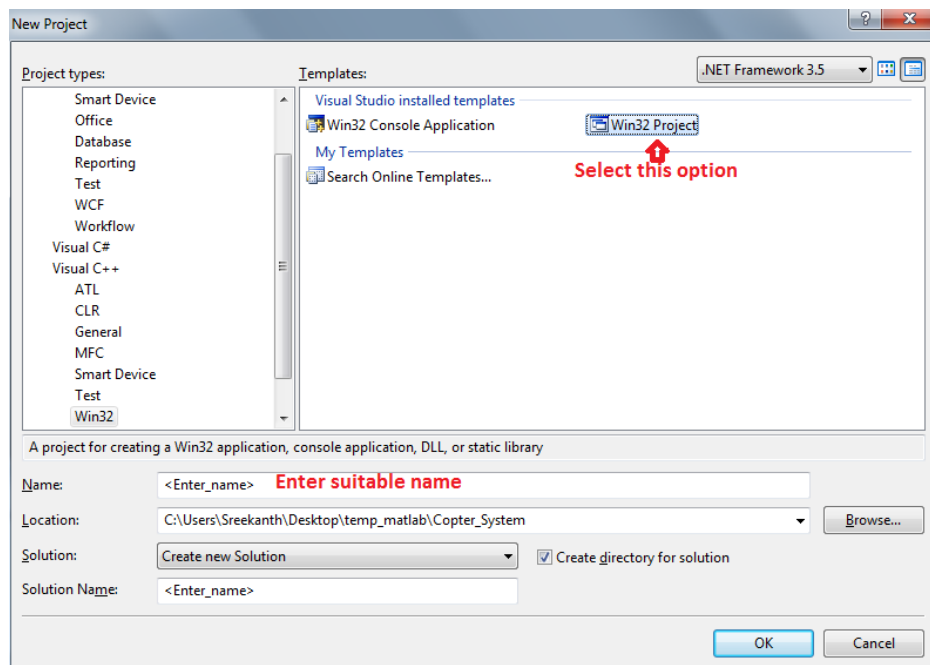
For more details on these subsystems, please see Appendix.

2.7 Visual Studio settings for compiling S-Functions

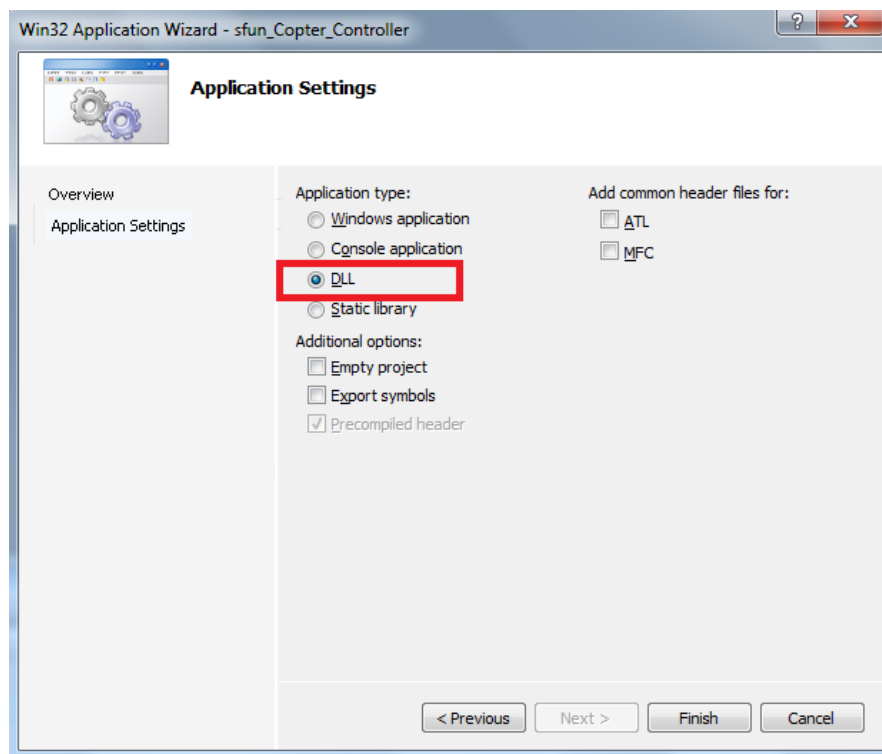
S-Functions are written & compiled into .dll files by Microsoft® Visual Studio, since it offers the possibility to debug the code by interfacing it with the MATLAB/Simulink®.

Here are the general settings to create the Visual Studio project & to compile the source code into .dll files.

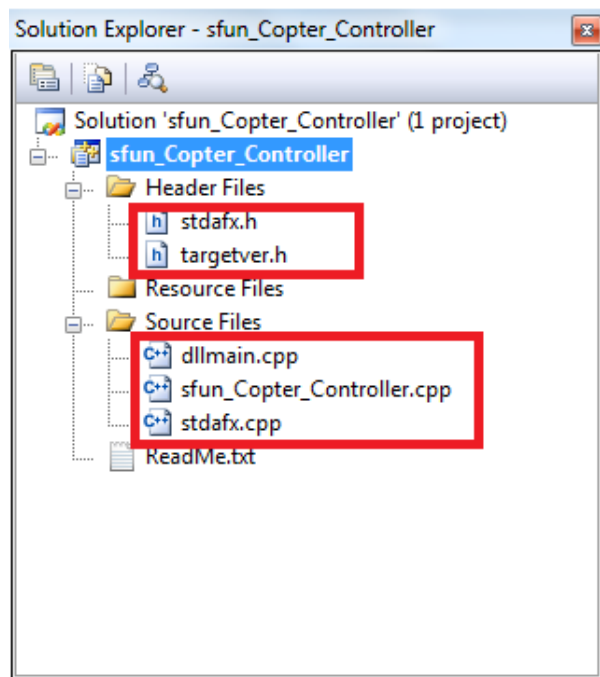
1. Create a new Win32 application & suitably name the project.



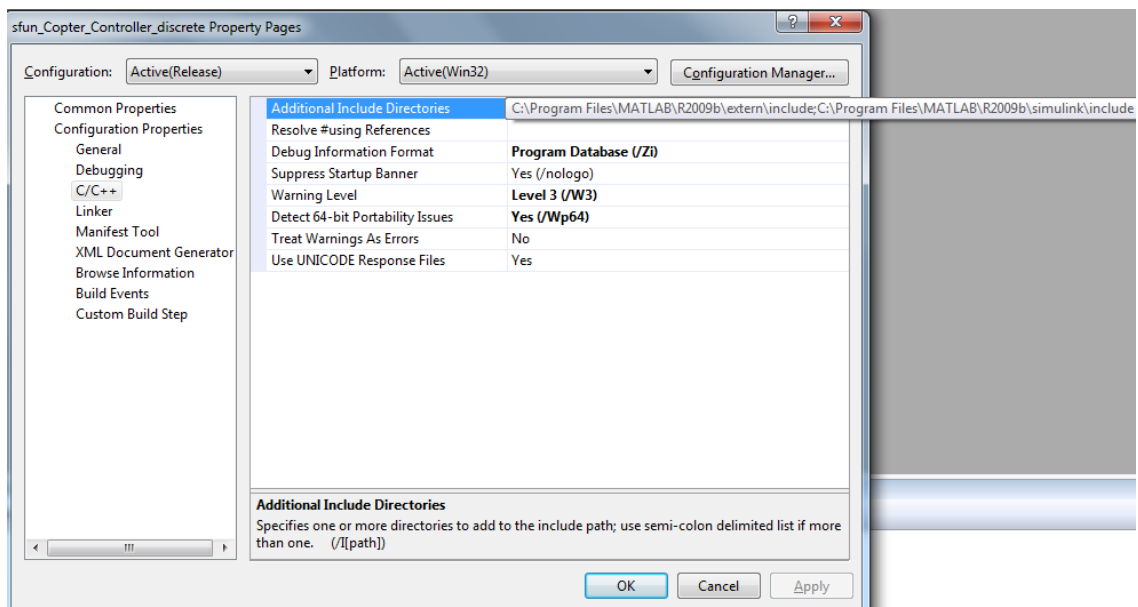
2. Choose the "Application Type" as DLL.



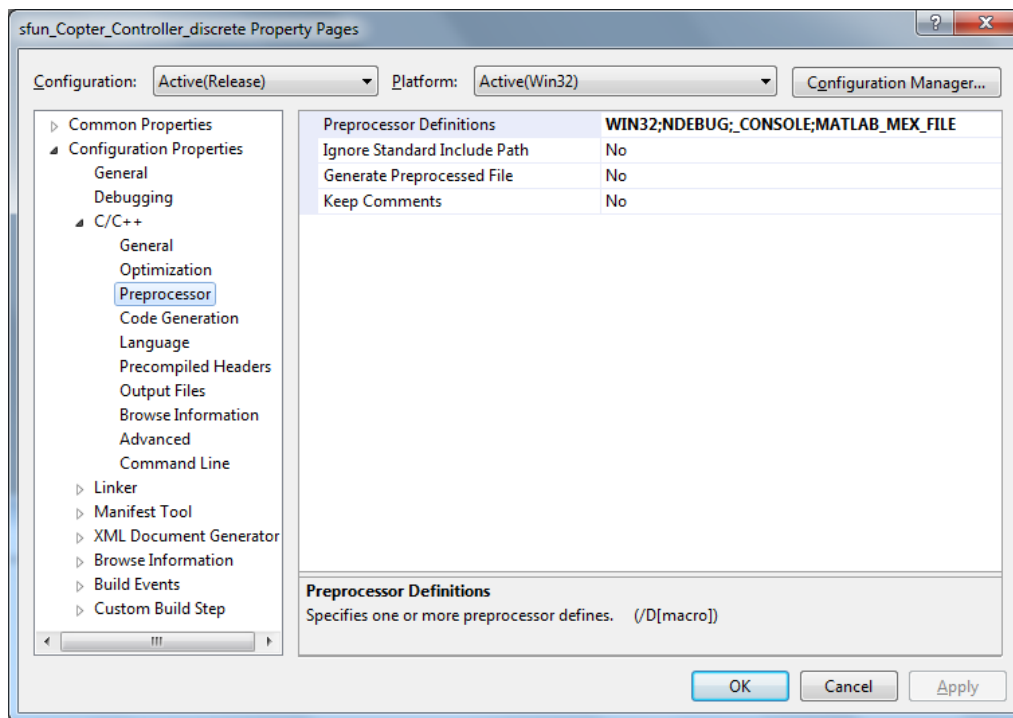
3. Remove these files & add your own source & header files for compilation.



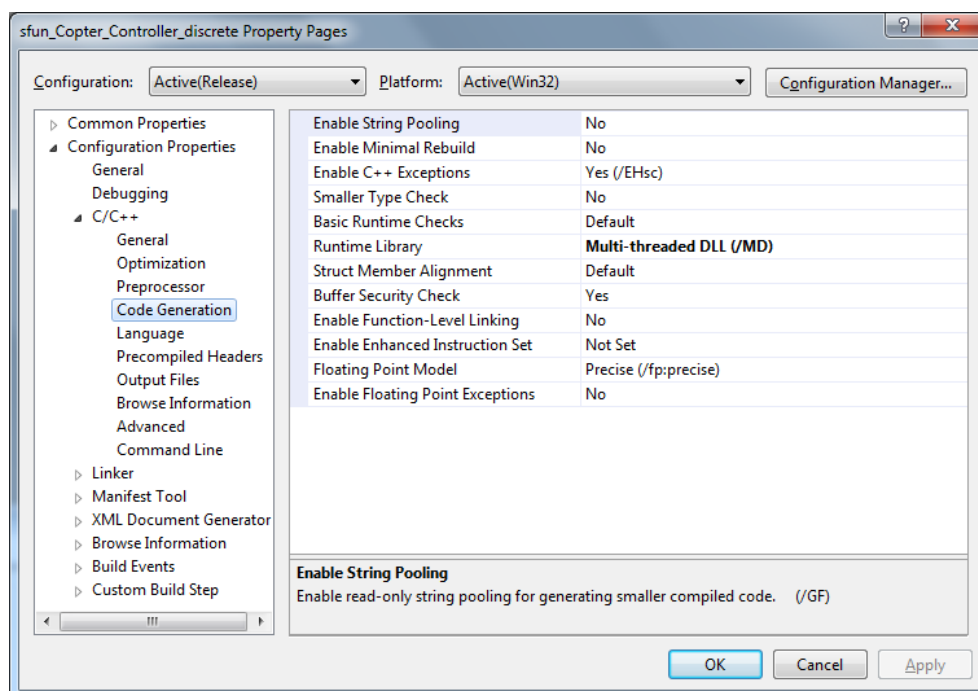
4. Select the highlighted options for the field "C/C++".



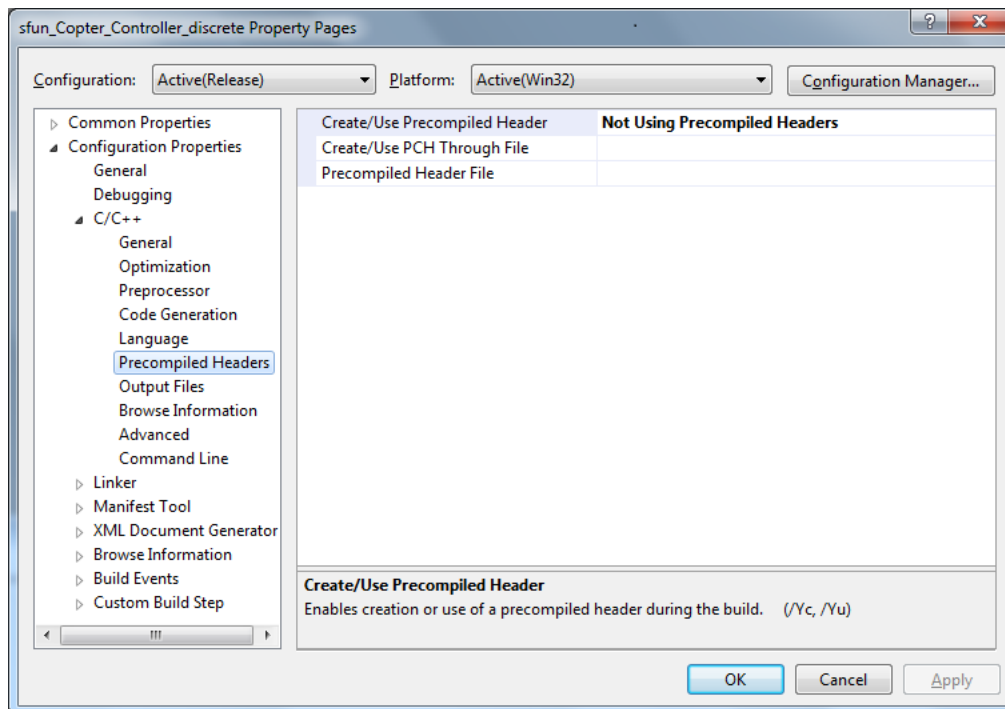
5. Select the highlighted options for the field "Preprocessor".



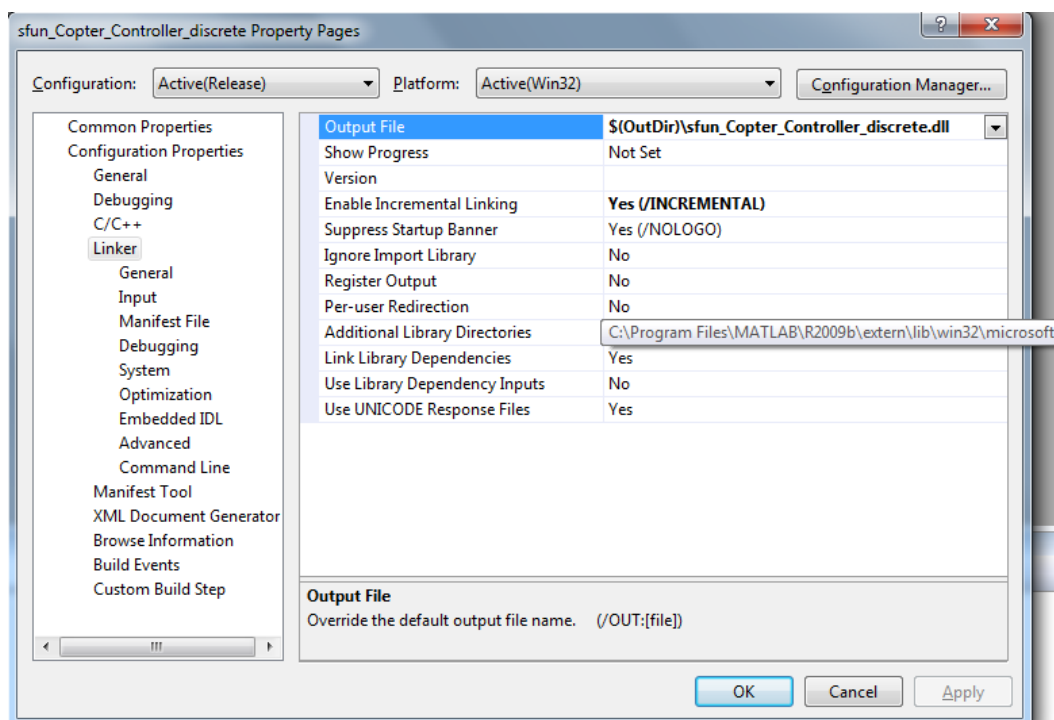
6. Select the highlighted options for the field "Code Generation".



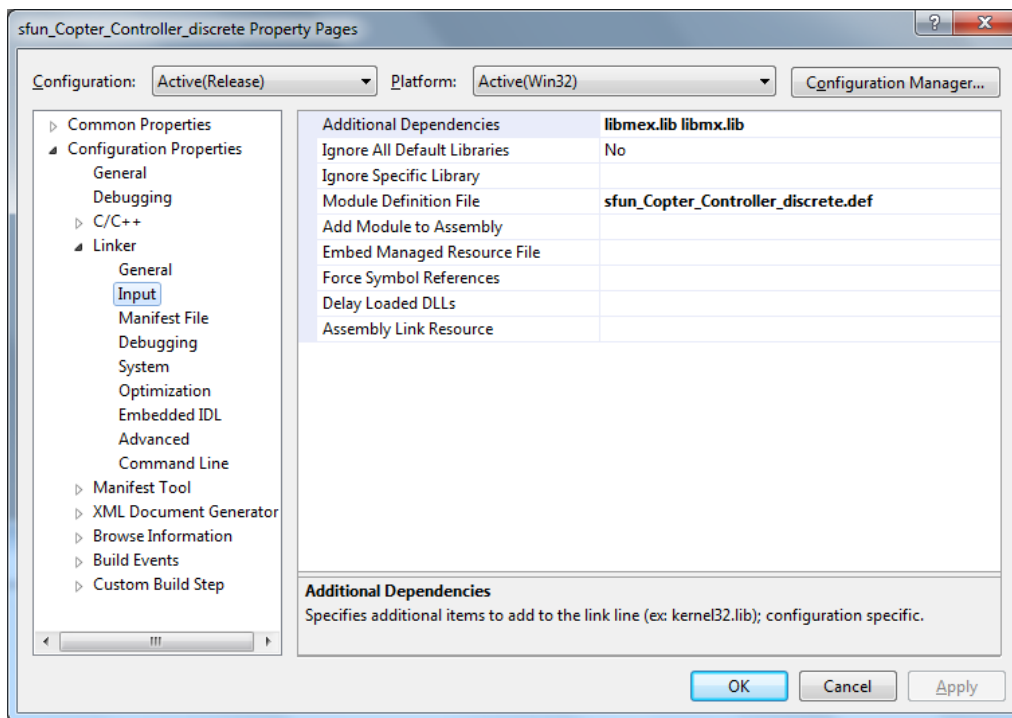
7. Select the highlighted options for the field "Precompiled Headers".



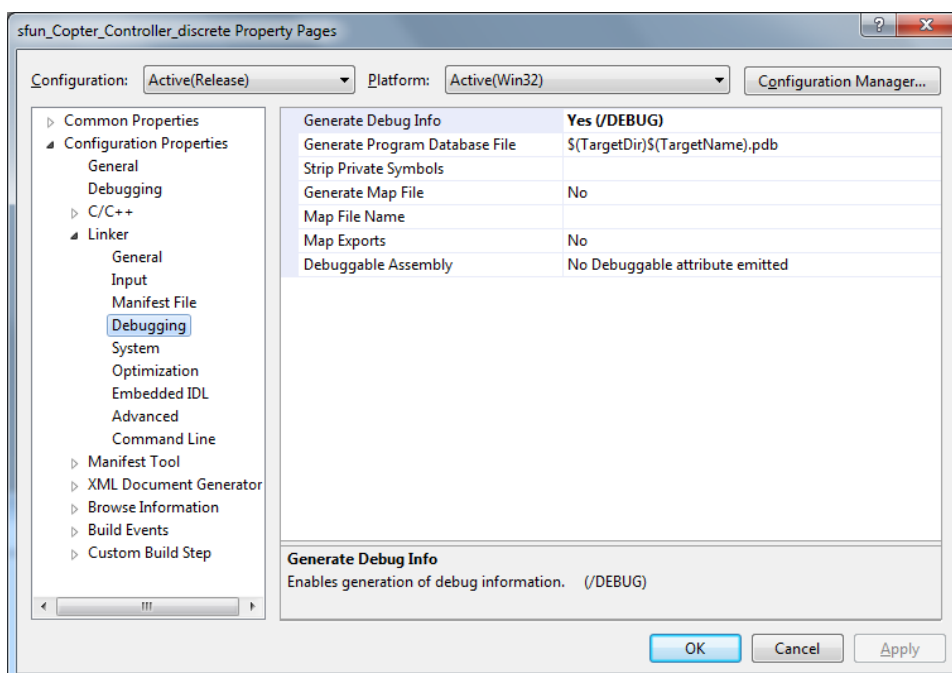
8. Select the highlighted options for the field "Linker".



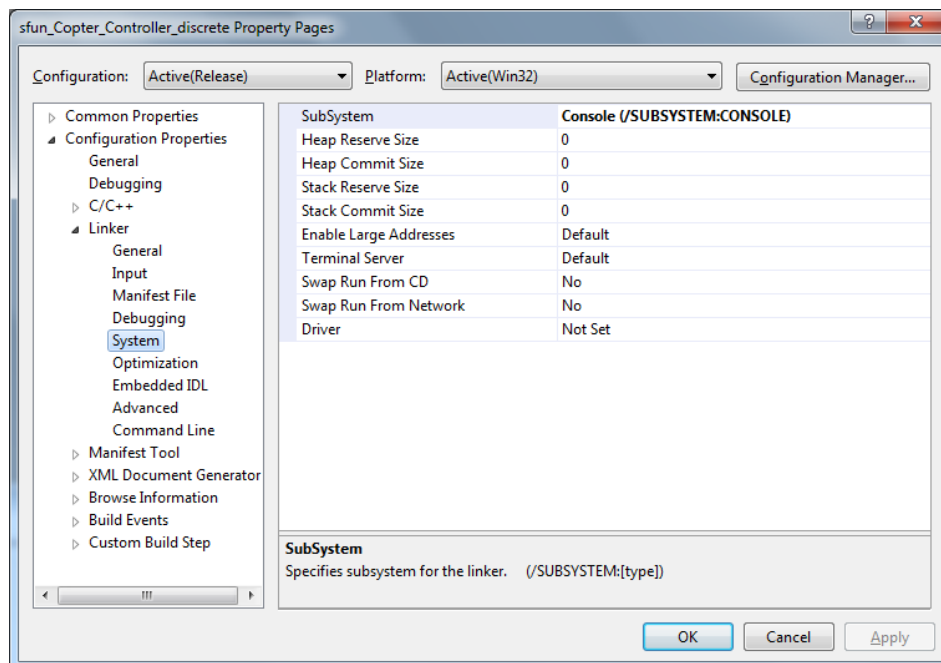
9. Select the highlighted options for the field "Input".



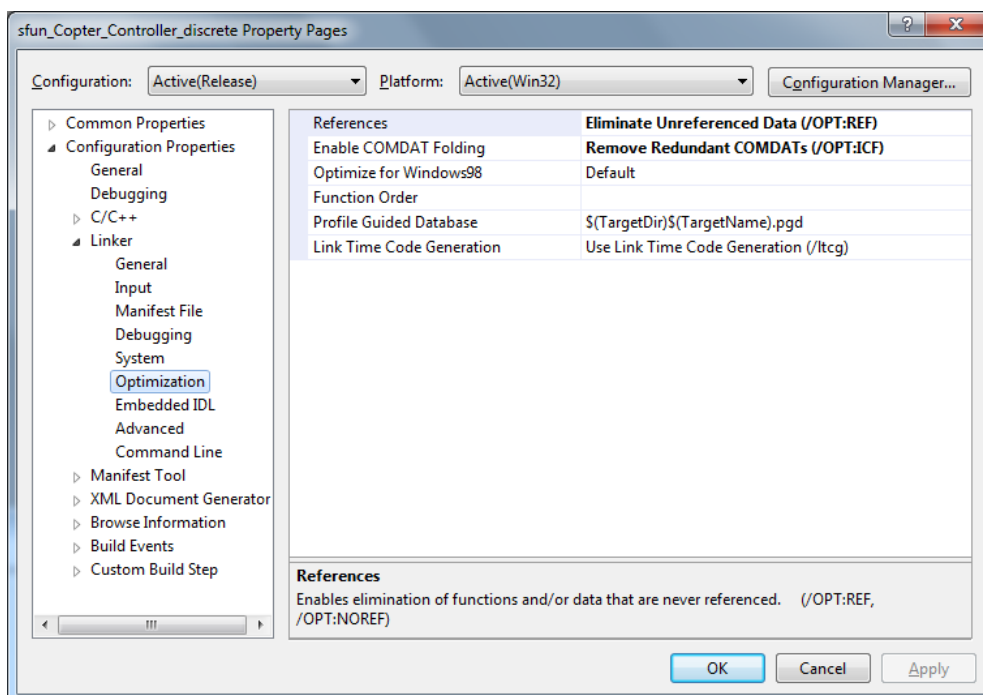
10. Select the highlighted options for the field "Debugging".



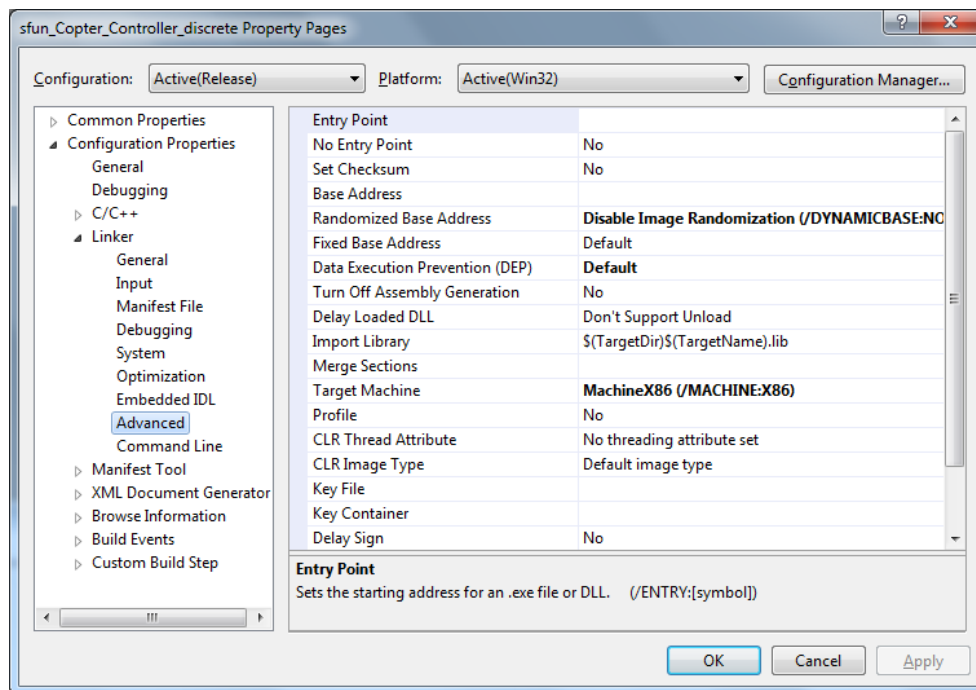
11. Select the highlighted options for the field "System".



12. Select the highlighted options for the field "Optimization".



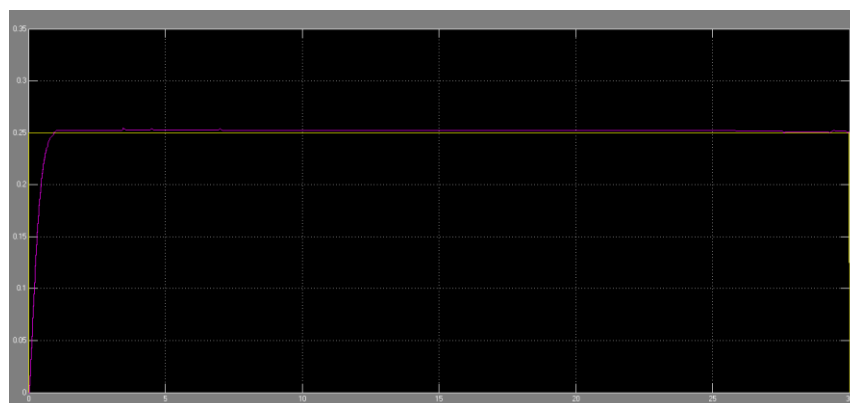
13. Select the highlighted options for the field "Advanced".



2.8 Test cases & Simulation results

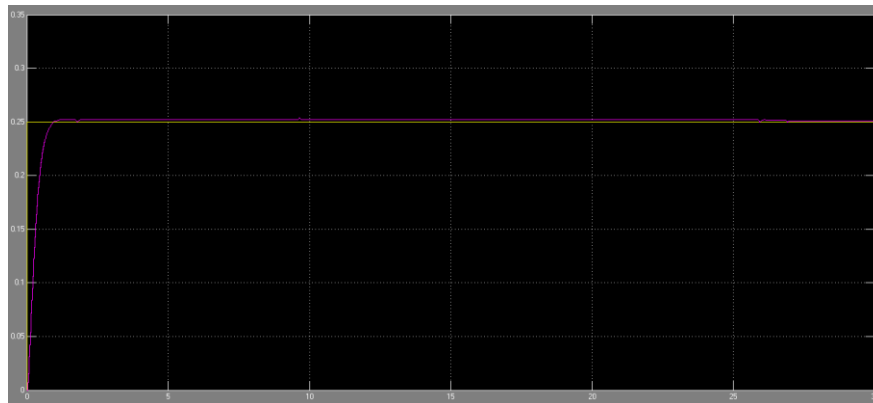
a) Step Roll angle simulation

Desired Roll angle = 0 to 0.25 rad step input in 0.01 sec



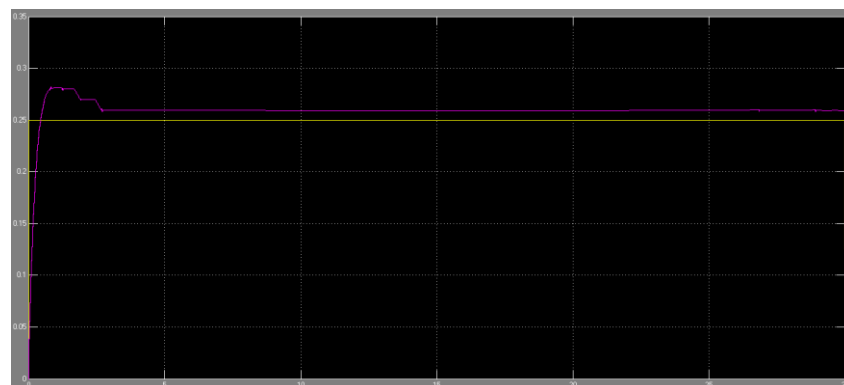
b) Step Pitch angle simulation

Desired Pitch angle = 0 to 0.25 rad step input in 0.01 sec



c) Step yaw velocity simulation

Desired yaw velocity = 0 to 0.25 rad/s step input in 0.01 sec



2.9 Improvements & Conclusions

The main goal of the project was to have a stable flight while hovering as well as during quick maneuvers. Hence we have chosen the PID controllers for the roll angle, pitch angle & yaw velocity stabilization. Even though the simulations showed satisfactory results with this type of controller, final result was not robust enough & the actual flight was not fully autonomous. Possible reasons could be:

- The simulated quadrotor model might be insufficient in addressing all the non-linearities (e.g. taking into account external disturbances like wind influence)
- There was no model to account for the temperature compensations for the sensors (accelerometer & gyros)
- To some extent we feel the sensor filter block implemented with "Complementary Filter" might not give a good estimate of the angles. Although one could argue that we can get reasonable values from such a filtering estimate, one could use more sophisticated estimating techniques like "Kalman Filter" instead of "Complementary Filter". But the mathematical overhead of such complicated estimating techniques sometimes far outweigh the actual benefits achieved from such methods.

Hence we feel the probable improvements would be in the area of accurate quadrotor dynamics modeling, designing the compensation blocks for the sensors (temperature) & to some extent incorporate a better filtering technique to get the angle estimates.

3 Quadrocopter Electronics

3.1 System Description

The electronic system of the quadrocopter mainly consists of the flight control unit, one brushless controller for each motor and the motors themselves. The flight control evaluates its sensors and the remote control, calculates the new setpoints and sends them to the brushless controllers. The brushless controllers home the power electronics and are responsible for the correct control of the brushless motors (3-phase control). Additionally the flight control is connected to a base station via a wireless connection to transmit the actual status and receive new parameter values. Figure 3.1 illustrates the architecture of the electronic system.

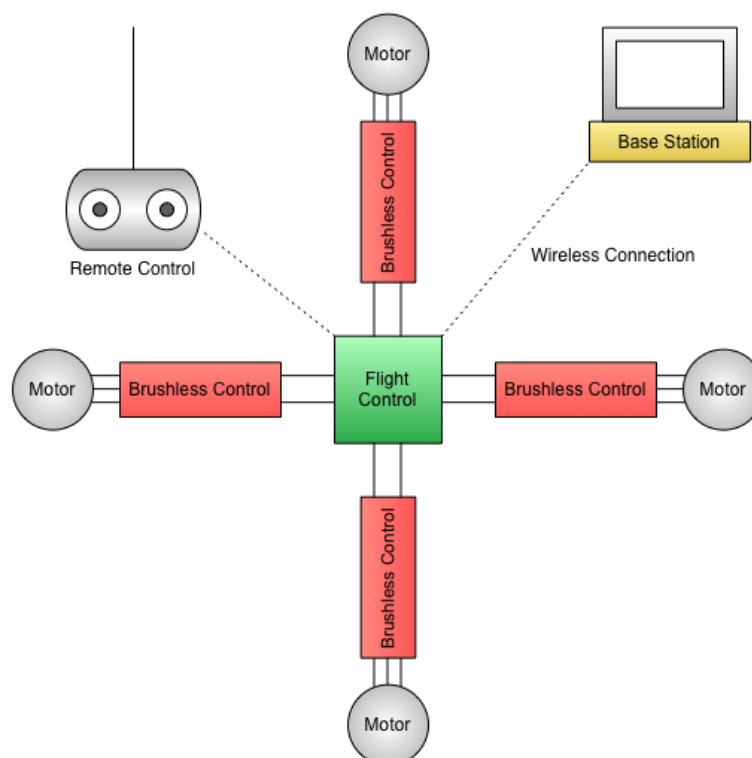


Figure 3.1: Electronic system of the quadrocopter

3.2 Central Control Unit

The central control unit ("flight control" in Figure 3.1) was developed by the HS-Esslingen (faculty electronics in Göppingen). The microcontroller, used for the flight control task and additional tasks, is an MC9S12XDT256 from Freescale. On the printed circuit board (PCB) there are all sensors mounted needed for the calculation of the actual position in space. This includes an accelerometer measuring the acceleration in x-, y- and z-direction, three gyroscopes measuring roll-, pitch- and yaw-velocity and an air pressure sensor to determine the height. A battery sensor (voltage divider) is needed for the calculation of the new setpoints for the brushless controllers because the resulting RPM/thrust of the motors depends on the actual battery voltage. A beeper and two LEDs are available for diagnosis. The XBee module for the communication with the base station is also directly mounted on the flight control PCB.

Besides that, there are a couple of external interfaces available. The interfaces for the remote control, the I²C bus to command the brushless controllers and the Background Debug Mode (BDM) programming interface are essential for correct operation. The interfaces for a distance sensor, a GPS (Global Positioning System) receiver and a servo (e.g. for a camera mounting) are available for future extensions. A second possibility to program the microcontroller is the dedicated Universal Asynchronous Receiver Transmitter (UART) interface, but it's not used at the moment.

Figure 3.3 shows all hardware interfaces of the central control unit. The complete schematic is in the appendix (corrected pin numbers are red).

CAUTION: There were some errors in the schematic and on the PCB, which were corrected during the project:

- Reference voltage of the microcontroller's analog to digital converter was not connected
- Level converter IC (3.3 V to 5 V) was wrong connected (wrong symbol in schematic)
- Short circuit between 3.3 V and ground (GND) if power supply cables were soldered

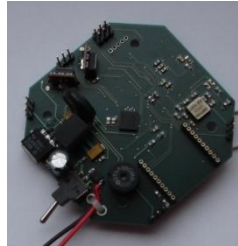


Figure 3.2: PCB of the central control unit

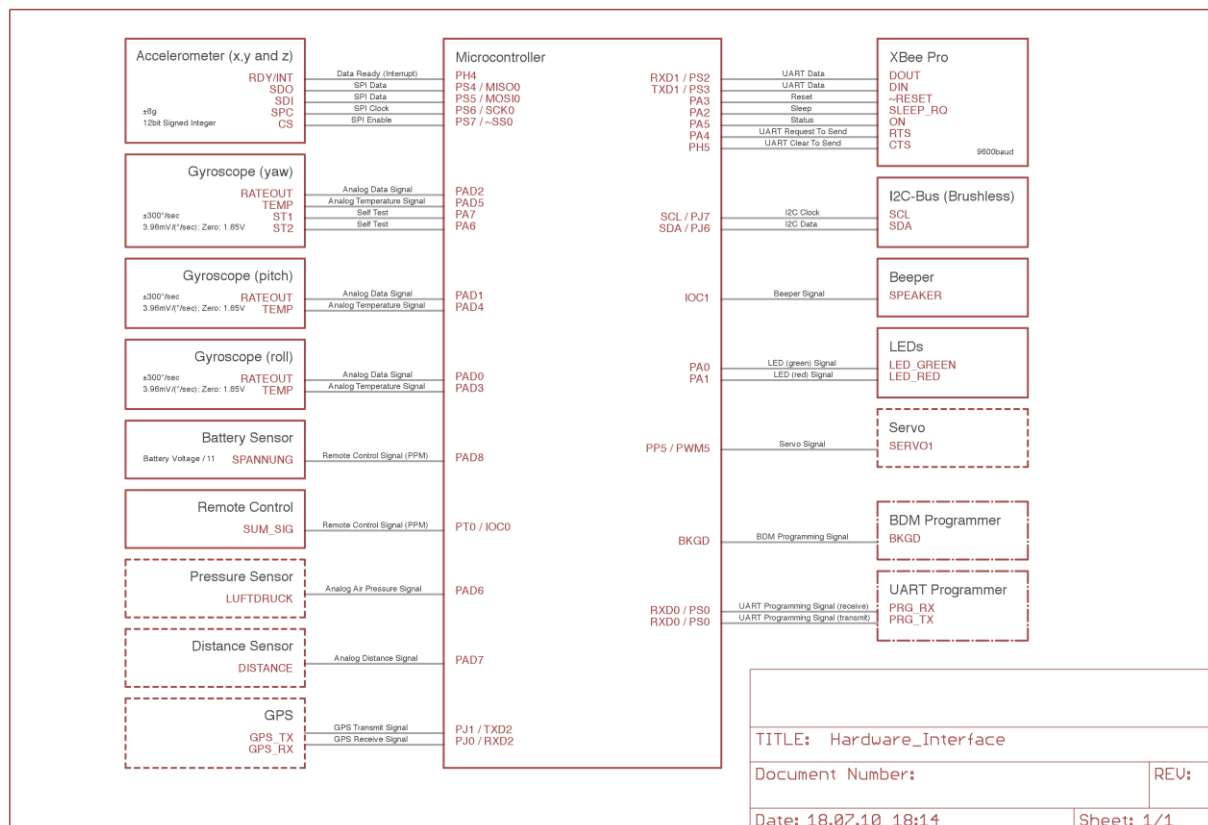


Figure 3.3: Hardware interfaces

3.3 Accelerometer

The accelerometer, used for the flight control, is an intelligent sensor measuring x-, y- and z-acceleration. It has a Serial Peripheral Interface (SPI) for configuration and value return. The sensor is configured to measure accelerations up to $\pm 6g$ and to deliver results as 12-bit signed integer values. In addition the accelerometer provides a data ready signal, which indicates when new sensor data is available.

SPI is a serial, byte-oriented, master-slave bus system using three or four wires. The flight control to accelerometer interface uses the version with four wires: MISO (Master In Slave Out), MOSI (Master Out Slave In), SCK (Serial Clock) and CS/SS (Chip Select/Slave Select). CS/SS is a dedicated line to each slave. Hereby the master (flight control) selects the slave (accelerometer) he wants to communicate with. To start a communication, the master pulls the CS/SS wire to "low". Afterwards he starts sending bytes. SPI forms a shift register between master and slave, so at the time when the master "clocks out" a byte he will simultaneously receive a byte from the slave. That means to read data from a slave, the master has to send dummy data (e.g. 0xFF). The bus clock is 12MHz.

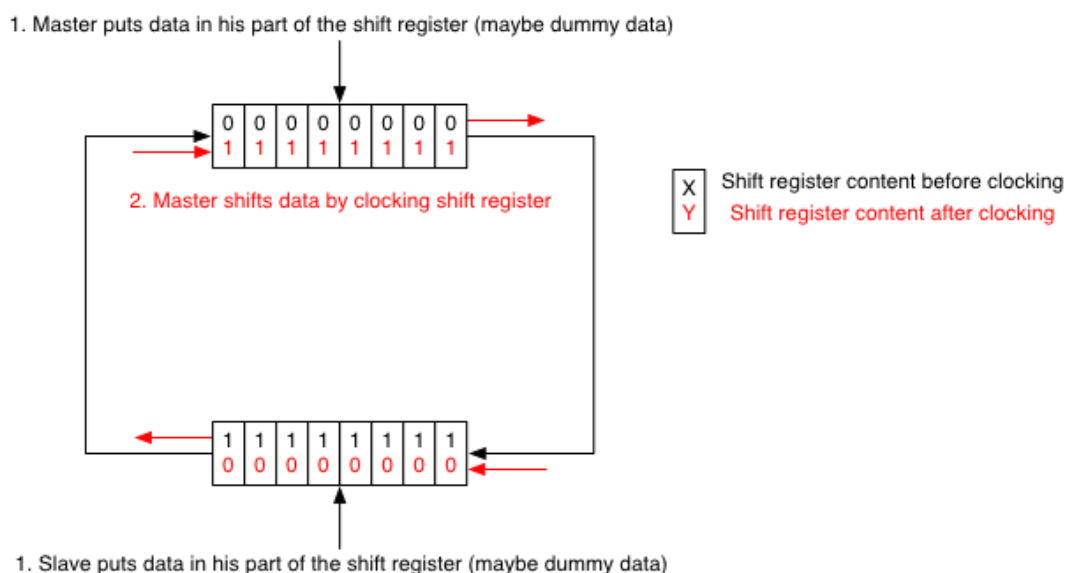


Figure 3.4: SPI principle

Following protocol is used for the communication between flight control and accelerometer: After the CS/SS wire has been put to "low" the flight control sends a command byte, which tells the accelerometer what to do. In case of configuring the sensor, the master sends afterwards the values for the configuration register(s). In case of requesting the actual sensor values, the accelerometer puts the data in its part of the shift register. If the master now shifts out dummy data, on the other side he will receive the sensor data. That means the master pulls down the CS/SS wire, sends the command byte for reading sensor values followed by six dummy bytes (two byte for each axis).

3.4 Gyroscopes / Temperature Sensor

There are three gyroscopes mounted on the flight control PCB. They measure the angular velocity of roll, pitch and yaw in a range of ± 300 °/sec. The result is available as analog voltage between ground and reference voltage (3.3 V). Half of the reference voltage corresponds to 0°/sec and the change rate is 3.96 mV/(°/sec). Additionally the gyroscopes contain a temperature sensor, which also provides an analog voltage as output signal. It is half of the reference voltage at 25 °C and changes with a rate of 5.94 mV/°C. For a first approximation a linear characteristic is assumed.

3.5 Battery Sensor

The battery sensor is a voltage divider between battery voltage and ground. The sensor voltage (measured between the two resistors) is 1/11 of the battery voltage.

3.6 Remote Control

A remote control receiver normally outputs a pulse-position modulated signal (PPM) for each channel. To save resources a special receiver is used (ACT DSL-4top (35 MHz) Typ: MK - www.mikrocopter.de).



Figure 3.5: Remote control receiver with sum-signal output

This receiver outputs the so-called sum-signal at the pin of the first channel. In the sum-signal there are all channels (up to nine) encoded. Figure 3.6 shows one frame of the sum-signal. The length of a low pulse represents the desired value of the remote control channel. For the untrimmed state, the low pulse duration is between 700 μsec and 1500 μsec (trimmed: 600 – 1600 μsec). A 400 μsec high pulse separates two channels. Two consecutive frames are distinguished by a long low period. Its absolute length is variable since one frame has a fixed length of 22msec. Even if all nine channels are at the maximum, the period is longer than the maximum low pulse duration of one channel and therefore this low period is used for synchronization.

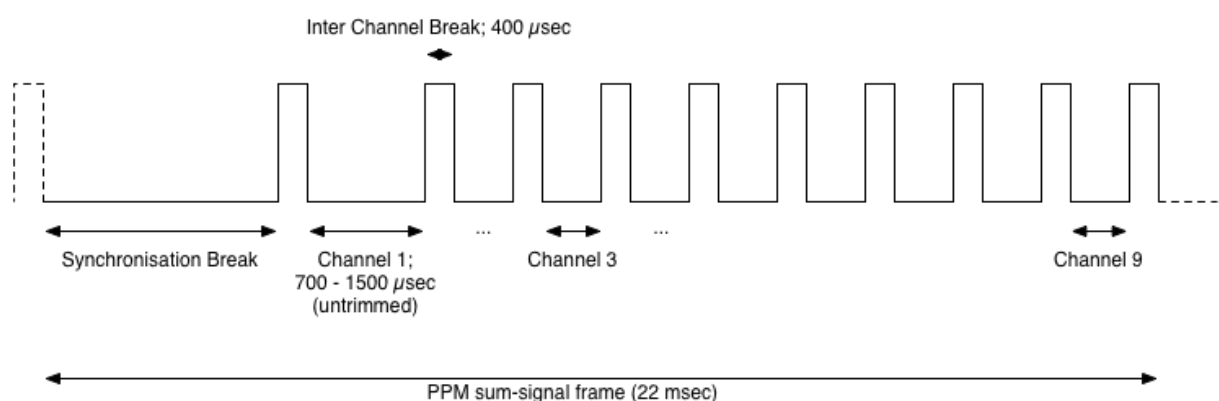


Figure 3.6: PPM sum-signal frame

3.7 XBee Pro

XBee Pro is a commercial solution for wireless communication using Universal Asynchronous Receiver Transmitter (UART). XBee is used to transmit actual status information from the quadrocopter to the base station and to send new parameter values from the base station to the quadrocopter. Therefore there are two XBee modules involved, one at the base station and one at the quadrocopter. The modules take the UART signal and transmit it wireless to the other side. There it is received and sent out again over the UART interface.

The XBee module at the quadrocopter is connected to the microcontroller over several wires, but only the two wires Receive Data (RxD) and Transmit Data (TxD) are used for data transmission. Additionally the Clear To Send (CTS) wire is used for flow control purposes. At the microcontroller the UART is named Serial Communication Interface (SCI).



Figure 3.7: XBee Pro modul

Because UART itself is a serial, byte oriented communication interface (e.g. RS-232) a special communication protocol is needed to transmit bigger data packets like status information. Therefore an own protocol is used. See chapter 5.

3.8 Brushless Controller

The brushless controllers are commercial parts (www.mikrokoetter.de). They control the connected brushless motor via Pulse-Width-Modulation (PWM) of the three phases.

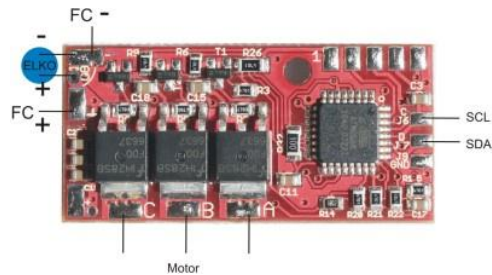


Figure 3.8: Brushless controller

The setpoint, sent by the flight control over an Inter Integrated Circuit (I^2C) bus, is the duty factor of the PWM. In a first approximation, the brushless motor can be regarded as a resistor. Therefore the setpoint corresponds to the average current flowing through the motor, which generates a certain torque. This results in RPM/thrust. Because the current drops with lower battery voltage ($I = U/R$) the setpoint does not represent RPM directly.

The possibility to read back data is currently not used.

Because all four brushless controllers are connected to one I^2C bus, there is the need to have addresses to send a setpoint to the right controller. The address of the brushless controller is specified by a solder jumper. To get no constant yawing torque, front and rear motor must turn in one direction, left and right motor must turn in the other direction. Because the address specifies the direction of turning, each controller must get a specific address. Table 3.1 shows the addresses of the brushless controllers.



Figure 3.9: Solder jumper on brushless controller PCB to select the address

Brushless Controller	Address	Jumper 1-2	Jumper 2-3
Front	0	open	open
Rear	1	open	closed
Right	2	closed	open
Left	3	closed	closed

Table 3.1: Brushless controller addresses

Setpoints are sent from the flight control to the brushless controllers via an I²C bus. The I²C bus is a serial, byte-oriented, master-slave bus using two wires: SCL (Serial Clock Line) and SDA (Serial Data Line).

The bus master, here the flight control, initiates all communication on the bus. After the start signal the master sends an address byte consisting of the seven-bit address of the slave and a read/write bit ("0" means master sends data to slave, "1" means master requests data from slave). The data bytes are then either sent by the master or by the slave. To transmit the setpoint to the brushless controllers, the master sends one data byte. The communication is finished by a stop signal. The brushless controllers require using a bus clock of 200kHz.

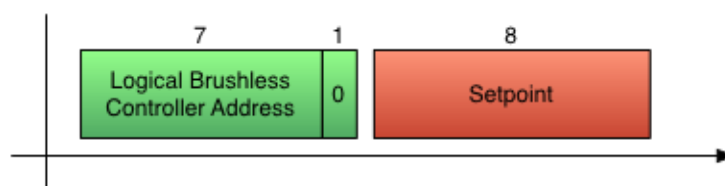


Figure 3.10: Transmission of new setpoint to a brushless controller

CAUTION: The brushless controller address, specified by the solder jumper, is not the logical address used for the I²C bus. See chapter 4.2.4.

3.9 Brushless Motor

The brushless motor / propeller combination, used for the quadrocopter (Robbe Roxxy 2827-34 / 10 x 4.5), have the setpoint to thrust characteristic shown in Figure 1.19. The dependency on the battery voltage is also depicted.



Figure 3.11: Robbe Roxxy brushless motor

3.10 Beeper

The beeper of the flight control is used for diagnostic purposes i.e. signaling low battery voltage. It is connected to a timer channel of the microcontroller, which generates the desired frequencies to drive the speaker.

3.11 Light Emitting Diodes (LEDs)

There are two LEDs mounted on the flight control PCB, a red and a green one. Both are used for diagnostic purposes. The green one is used as "alive" indication blinking with 0.5 Hz. The red one is on, if no correct remote control signal is received (receiver sends no valid PPM sum-signal frame).

3.12 Other Components

Air pressure sensor, distance sensor, Global Positioning System (GPS), a servo (e.g. for camera) and a UART programmer are currently not used. The central microcontroller is programmed with the BDM programmer from PEmicro.



Figure 3.12: BDM programmer from PEmicro

4 Embedded Software

4.1 General Structure

The embedded software, running on the HCS12X microcontroller, is divided in three layers: The Application Layer, the Data Layer (containing the real-time database) and the Hardware Abstraction Layer (HAL). The structure and the modules of each layer are shown in Figure 4.1. On Application Layer, the embedded software follows the time-triggered approach using time-triggered "tasks". Modules, handling asynchronous processes (*QH_remote* and *QH_xbee*), use interrupts. This is done to keep the efficiency of the software.

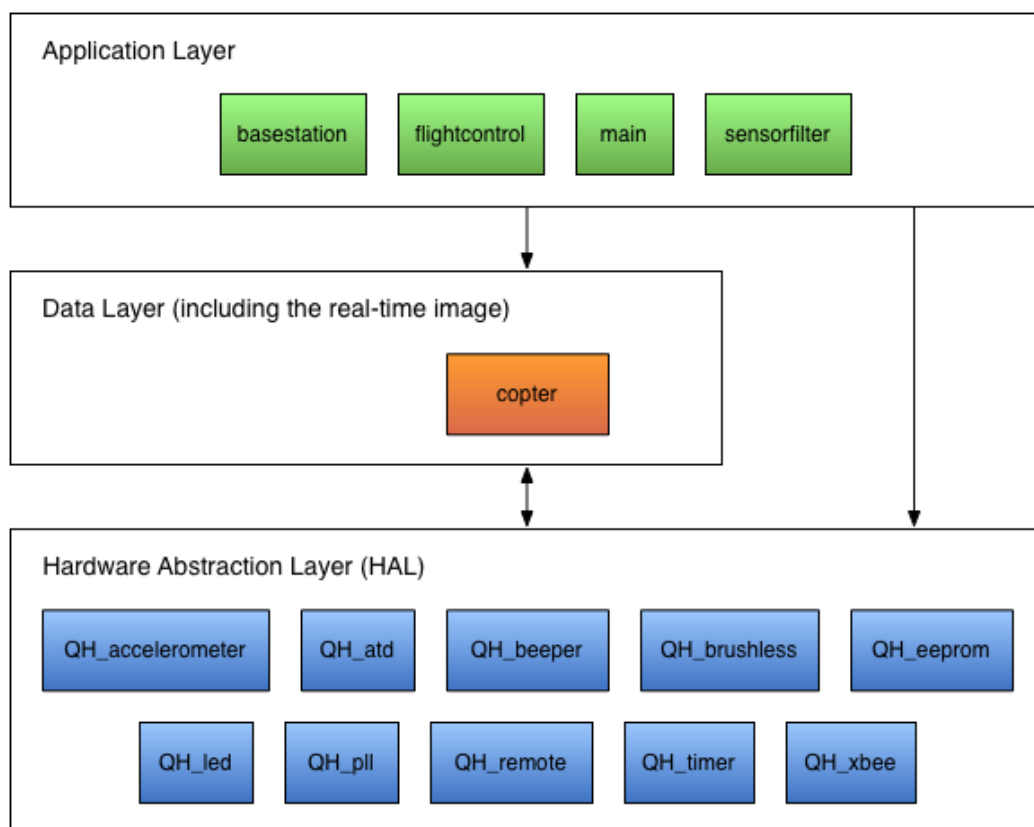


Figure 4.1: Structure of the embedded software

The Hardware Abstraction Layer (HAL) encapsulates all functions, which are directly related to the microcontroller, sensors and actuators. This implies that in the layers above, there are no CPU registers used directly.

In the embedded software, there are the data types used listed in Table 4.1. There is no use of floating-point numbers. Therefore all types are integer based. To be flexible, the data types are defined in the header file *typedef.h*. The definitions are specific for each platform on which the software runs, because e.g. an *int* variable on the embedded platform (16 bit) has a different value range than an *int* variable on a personal computer (32 bit). To compensate this and to achieve the same software behavior, the appropriate type definitions have to be used. This is especially important for the modules *flightcontrol* and *sensorfilter* because the same software is used on a personal computer for simulation purposes and within the embedded system. Without such type definitions at one place, all variables etc. must be changed in order to get the same software behavior on different target platforms.

Name	Sign	Width [Bit]	Value range
bool	(unsigned)	1 (8)	TRUE or FALSE
uint8	unsigned	8	0 to 255
int8	signed	8	-128 to 127
uint16	unsigned	16	0 to 65535
int16	signed	16	-32768 to 32767
uint32	unsigned	32	0 to 4294967296
int32	signed	32	-2147483648 to 2147483647

Table 4.1: Data types

In the following sections the software is described in detail. However this is done on an abstract view. The C-code itself is well documented and provides a low level documentation.

4.2 Hardware Abstraction Layer (HAL)

The Hardware Abstraction Layer consists out of ten software modules. One module encapsulates/abstracts a part of the microcontroller, a sensor or an actuator. The naming convention of the modules is QH_<device>. QH stands for Quadcopter HAL.

CAUTION: Every module contains an `...Init()` function, which must be called before any other function of this module. Additionally the sequence is important in which the `...Init()` functions are executed. `pllInit()` must be called first, followed by `timerInit()` because other modules require a working timer unit. E.g. QH_accelerometer needs a working SPI to configure the sensor. To derive the SPI clock, a configured timer unit is necessary.

4.2.1 QH_accelerometer

This module holds all functions required to handle the intelligent accelerometer. The sensor is connected to the microcontroller via Serial Peripheral Interface (SPI). The values, received from the sensor, are converted into physical ones [$10 \cdot \text{m/sec}^2$] for internal usage without floating point unit.

The software module provides the following functions to the outside:

```
void accelInit(void)
void accelCalibrate(void)
void accelGetValues(void)
```

After initializing the SPI interface (SPI0 of the microcontroller) and the sensor itself by the `accelInit()` function, the accelerometer should be calibrated. The zero level of all three axis (X, Y and Z) is defined by using `accelCalibrate()`. Therefore the copter must stand on a plain ground and it must be in rest. The function `accelGetValues()` writes back the actual sensor values (raw and physical) to the `copterState` structure (real-time database, see chapter 4.3.2).

HINT: The data ready signal is currently not used, because it caused infinite waiting times once in a while.

HINT: The accelerometer provides a self-test mode. In future it can be used to check, if the accelerometer works fine and additionally to enhance the physical value calculation (calibration purposes).

4.2.2 QH_atd (Analog to Digital Converter)

This module holds all functions required to use the built in analog to digital converter of the microcontroller. It has a resolution of 10 bit. There are different types of analog sensors connected. First of all the three gyroscopes for measuring roll, pitch and yaw velocity. Additionally a temperature sensor (contained in a gyroscope), an air pressure sensor and a voltage divider, measuring the battery voltage, are connected. The values of the A/D converter are converted into physical ones for internal usage without floating point unit – [100*rad/sec] for roll, pitch and yaw velocity, [°C] for the temperature and [10*V] for the battery voltage. The air pressure and temperature sensor are currently not used.

The software module provides the following functions to the outside:

```
void atdInit(void)
void atdCalibrate(void)
void atdGetValues(void)
```

After initializing the analog to digital converter units (ATD0 and ATD1 of the microcontroller) by the `atdInit()` function, the gyroscopes should be calibrated. The zero level of all three gyroscopes is defined by using `atdCalibrate()`. Therefore the copter must stand on a plain ground and it must be in rest. The function `atdGetValues()` writes back the actual sensor values (raw and physical) to the `copterState` structure (real-time database, see chapter 4.3.2).

HINT: The gyroscopes provide a self-test mode. In future it can be used to check, if the gyroscopes work fine and additionally to enhance the physical value calculation (calibration purposes).

4.2.3 QH_beeper

This module holds all functions required to use the beeper. The beeper is connected to a timer channel of the microcontroller generating the pulsed signal for the beeper.

The software module provides the following functions to the outside:

```
void beeperInit(void)
```

```
void beeperOn(uint16 freq)
```

```
void beeperOff(void)
```

After the function `beeperInit()` has configured the timer channel (IOC1 of the microcontroller), `beeperon()` can be used to turn the beeper on. The beep frequency is specified by the parameter between 50Hz (50) and 10kHz (10000). The function `beeperoff()` turns the beeper off.

4.2.4 QH_brushless

This module holds all functions to control the brushless controllers. There is one brushless controller for each motor but all controllers are connected to the microcontroller via the same I²C bus. Therefore they have unique addresses (zero to three) defined by solder jumpers. For writing data to the brushless controllers, the address bytes of Table 4.2 must be used. The base is 0x52. On its top two times the specified brushless controller address is added. For reading data out of the brushless controller, the base is 0x53. This feature is currently not used.

Brushless Controller	Address	I ² C address byte (writing)
Front	0	0x52
Rear	1	0x54
Right	2	0x56
Left	3	0x58

Table 4.2: Brushless controller addresses

The transmitted setpoint (one byte) corresponds to a specific RPM depended on the battery voltage. A setpoint of zero means zero RPM (motor off) and 255 demands the maximal possible RPM. To transmit a new setpoint to a brushless controller, first the address byte and afterwards the setpoint byte have to be sent.

The software module provides the following functions to the outside:

```
void brushlessInit(void)
```

```
uint8 brushlessSetRpm(uint8 motorAddress, uint8 setpoint)
```

`brushlessInit()` configures the I²C bus (IIC0 of the microcontroller). By using `brushlessSetRpm()` a new setpoint is sent to a brushless controller. The first parameter specifies which brushless controller is addressed and the second one is the new setpoint.

4.2.5 QH_eeprom

This module holds all functions to store data (the quadrocopter configuration, `copterconfig`) in the Electrical Erasable Programmable Read Only Memory (EEPROM) of the microcontroller.

The software module provides the following functions to the outside:

```
void eepromWrite(void* ramAdr, void* eepromAdr, uint16 n)
```

```
void eepromRead(void* ramAdr, void* eepromAdr, uint16 n)
```

```
bool eepromIsErased(void* eepromAdr)
```

`eepromWrite()` copies the specified number of bytes (`n`) from the RAM to the EEPROM, whereas `eepromRead()` does it the other way round. `ramAdr` and `eepromAdr` specify the corresponding starting addresses. The function `eepromIsErased()` checks if the complete EEPROM memory is in erased state.

HINT: To be able to store data in the EEPROM, the file *Linker.prm* was modified. Please see C-code documentation.

4.2.6 QH_led

This module holds all functions to control the two LEDs mounted on the central PCB. They are connected to two general-purpose input/output pins.

The software module provides the following functions to the outside:

```
void ledInit(void)
uint8 ledOn(uint8 led)
uint8 ledOff(uint8 led)
uint8 ledToggle(uint8 led)
```

After the function `ledInit()` has configured the input/output pins (PA0 and PA1 of the microcontroller), `ledOn()` can be used to turn the LEDs on. The function `ledOff()` turns the LEDs off and `ledToggle()` toggles them. Which of the LEDs (green or/and red) is addressed by the functions can be specified by the parameter.

4.2.7 QH_pll

This module contains only one function, which configures the Phase Locked Loop (PLL) circuit to generate a bus frequency of 24Mhz out of the 4MHz oscillator clock.

The software module provides the following function to the outside:

```
void pllInit(void)
```

`pllInit()` must be called before any other `...Init()` function.

4.2.8 QH_remote

This module holds all functions to get the desired values from the remote control. The radio receiver is connected to a timer channel of the microcontroller measuring the pulse duration of each signal (channel) within the PPM frame (sum-signal). Each channel is decoded into a value between zero and 255.

Because the PPM frames arrive asynchronous to the program, the received channel values are double buffered. That means there is one buffer for reading and one buffer for writing values. During a new frame is received, the channel values are written into the write buffer. After the frame was received completely, they become valid and the buffers are exchanged. The new values are now present in the read buffer and written back if the actual remote values are requested. This principle is shown in Figure 4.2. For the reception itself a timer channel interrupt is used.

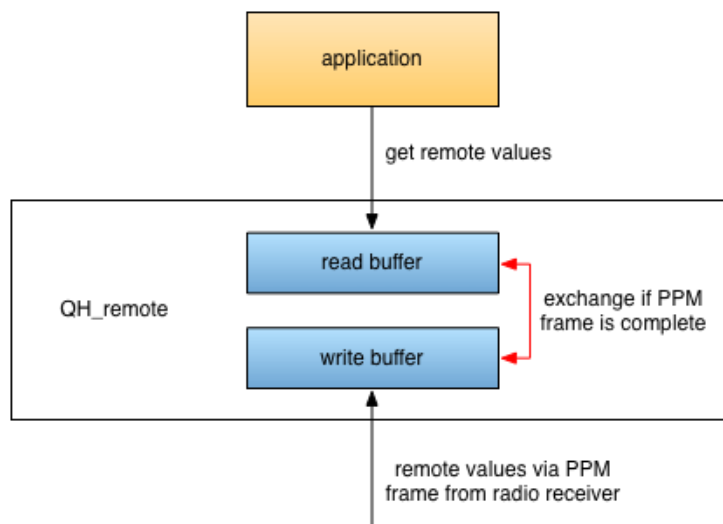


Figure 4.2: Double buffering of remote values

Because of the double buffer principle, the remote looks to the outside like a normal sensor, which can be polled.

The software module provides the following functions to the outside:

```
void remoteInit(void)
```

```
void remoteGetValues(void)
```

`remoteInit()` configures the timer channel (IOC0 of the microcontroller) to capture the pulse duration of the single channels within a PPM frame. Currently the first four channels are used. They are assigned to pitch-angle, roll-angle, thrust and yaw-velocity.

The function `remoteGetValues()` writes back the remote values to the `copterstate` structure (real-time database, see chapter 4.3.2). Zero corresponds to the minimal desirable value and 255 to the maximum desirable value. The conversion into physical values is done in the `copterActualizeState()` function of the `copter` module because the physical limits, e.g. the maximum desirable pitch angle, are specified in the `copter` configuration (`CopterConfig`, see chapter 4.3.1).

4.2.9 QH_timer

This module contains the function to configure the timer unit and a mechanism to schedule the time-triggered tasks of the Application Layer. This mechanism uses a timer channel and eight bits to indicate when predefined time intervals have elapsed. Each bit is set in a dedicated period and reset when it was read as set. Figure 4.3 shows the bits and their assigned time intervals.

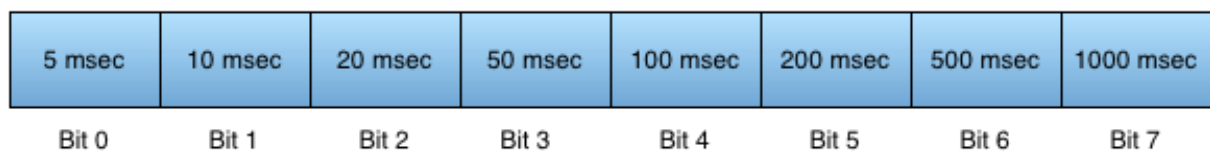


Figure 4.3: Bits indicating if a time interval has elapsed

The software module provides the following functions to the outside:

```
void timerInit(void)
```

```
bool timerIsFlagSet(uint8 timerFlag)
```

`timerInit()` configures the timer unit to have a tick period of one microsecond. This is essential to many other modules. Therefore this function must be called directly after `pllInit()` and before any other `...Init()` function. Additionally a timer channel (IOC7 of the microcontroller) is configured for the elapsed time interval indication. The function `timerIsFlagSet()` checks, if the time interval, specified by the parameter, has elapsed since the last check.

4.2.10 QH_xbee

This module holds all functions to use the XBee radio module to communicate with the base station. A Serial Communication Interface (SCI) connects the XBee module with the microcontroller. Because SCI is a byte oriented connection, a special communication protocol is needed. For details, please see chapter 5. The communication is asynchronous to the program. Therefore a similar approach as within the *QH_remote* module is used. Incoming data is double buffered. Received data becomes visible to the outside, if and only if a frame was received completely and the checksum was computed correctly. A difference to the *QH_remote* module is that a message is consumed by reading it. Also the read and write buffers must not be exchanged during a message is read by the application. Otherwise it would be possible that the application reads the first part of a message, then the buffers are exchanged due to a newly received message, and afterwards the application would read the remaining part. Figure 4.4 depicts this mechanism. For the reception itself the receive interrupt of the microcontroller's SCI interface is used.

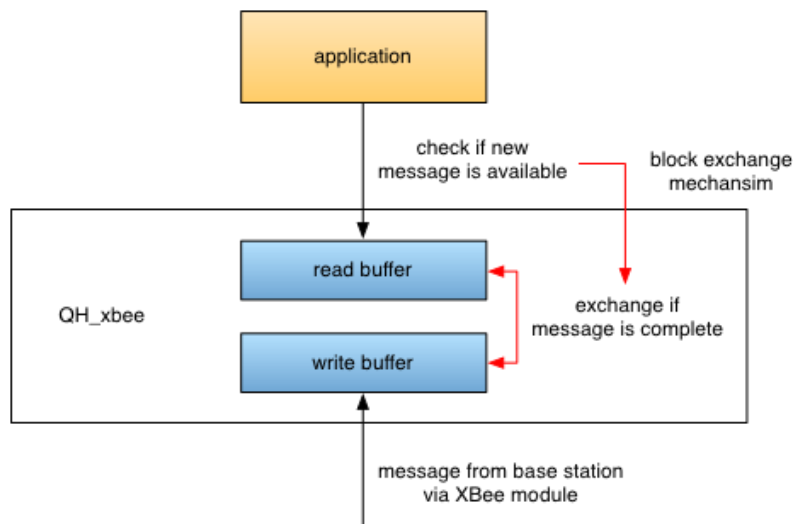


Figure 4.4: Xbee double buffering and buffer exchange block mechanism

The software module provides the following functions to the outside:

```
void xbeeInit(void)
uint8 xbeeSendMsg(XbeeMsg* msg)
uint8 xbeeCheckForMsg(XbeeMsg* msg)
```

After the function `xbeeInit()` has configured the SCI (SCI0 of the microcontroller), a message (`msg`) can be sent to the base station by using `xbeeSendMsg()`. `xbeeCheckForMsg()` checks if there is a new message available in the read buffer. If so, it is written back to the message structure passed as parameter.

4.3 Data Layer

This layer works as interface between the application (e.g. flight control) and the hardware. For this purpose, there are two big data structures used – `copterConfig` and `copterState`. Figure 4.5 depicts which modules access them. Changing the calibration parameters is critical and is therefore only possible at startup or when the motors are off.

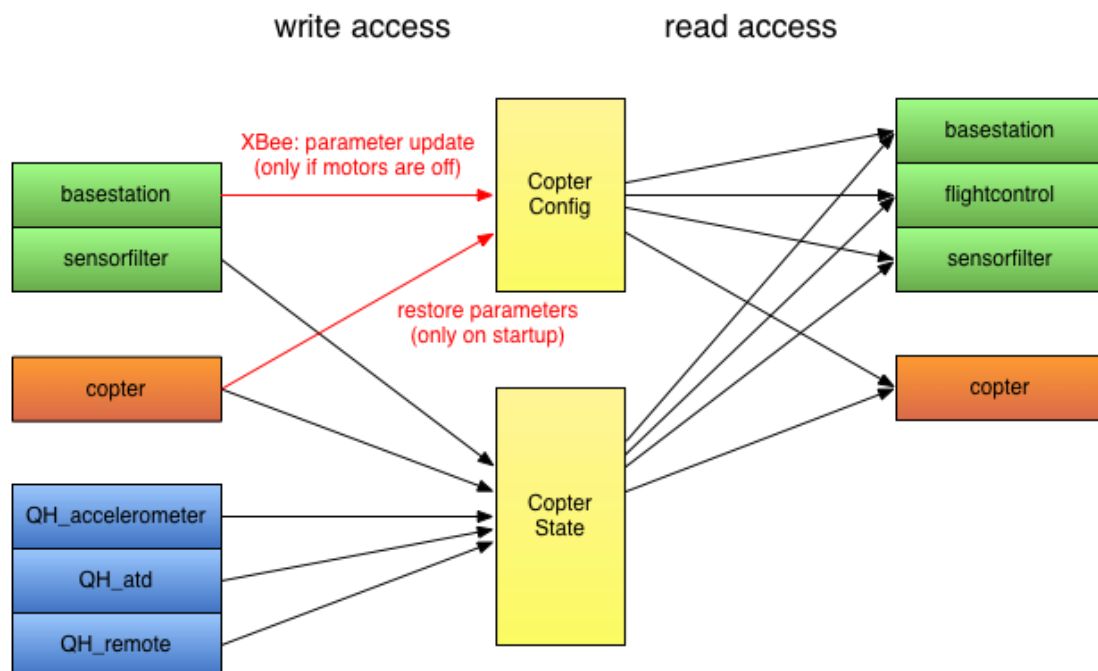


Figure 4.5: Software modules, accessing the data structures

4.3.1 Calibration parameter (CopterConfig)

The data structure `copterConfig` holds all important parameters of the system. Table 4.3 shows its layout. The parameters can be changed by the base station, but only if the motors are off.

Parameter	Abbreviation	Type	Range / [Unit]
P-value for roll control	kpR	int16	[100/sec ²]
I-value for roll control	kiR	int16	[100/sec ³]
D-value for roll control	kdR	int16	[100/sec]
P-value for pitch control	kpP	int16	[100/sec ²]
I-value for pitch control	kiP	int16	[100/sec ³]
D-value for pitch control	kdP	int16	[100/sec]
P-value for yaw control	kpY	int16	[100/sec]
I-value for yaw control	kiY	int16	[100/sec ²]
D-value for yaw control	kdY	int16	[100]
Low pass filter co-efficient (roll angle)	lpR	int16	0..100
High pass filter co-efficient (roll angle)	hpR	int16	0..100
Low pass filter co-efficient (pitch angle)	lpP	int16	0..100
High pass filter co-efficient (pitch angle)	hpP	int16	0..100
Inertia around the x-axis	inertiaX	int16	[1000*N*m*sec ²]
Inertia around the y-axis	inertiaY	int16	[1000*N*m*sec ²]
Inertia around the z-axis	inertiaZ	int16	[1000*N*m*sec ²]
Length of the booms	lenBoom	int16	[mm]
Maximum desirable force	forceMax	int16	[1000*N]
Maximum desirable yaw velocity	angVelYMax	int16	[(1000*rad)/sec]
Maximum desirable pitch angle	angPMax	int16	[1000*rad]
Maximum desirable roll angle	angRMax	int16	[1000*rad]

Table 4.3: Calibration parameter

4.3.2 System states (CopterState)

copterstate represents the actual system status. The table below shows the layout of the data structure. The raw values are only stored for debugging purposes.

State	Abbreviation	Type	Range / [Unit]
Acceleration in X direction (sensor)	accXRaw	int16	-2048..2047
Acceleration in X direction (physical)	accX	int16	$[(10*m)/sec^2]$
Acceleration in Y direction (sensor)	accYRaw	int16	-2048..2047
Acceleration in Y direction (physical)	accY	int16	$[(10*m)/sec^2]$
Acceleration in Z direction (sensor)	accZRaw	int16	-2048..2047
Acceleration in Z direction (physical)	accZ	int16	$[(10*m)/sec^2]$
Roll velocity (sensor)	angVelRRaw	int16	0..1024
Roll velocity (physical)	angVelR	int16	$[(100*rad)/sec]$
Roll angle (physical)	angR	int16	$[10000*rad]$
Pitch velocity (sensor)	angVelPRaw	int16	0..1024
Pitch velocity (physical)	angVelP	int16	$[(100*rad)/sec]$
Pitch angle (physical)	angP	int16	$[10000*rad]$
Yaw velocity (sensor)	angVelYRaw	int16	0..1024
Yaw velocity (physical)	angVelY	int16	$[(100*rad)/sec]$
Temperature (sensor)	tempRaw	int16	0..1024
Temperature (physical)	temp	int16	$[^{\circ}C]$
Air pressure (sensor)	airPressureRaw	int16	0..1024
Battery voltage (sensor)	batteryRaw	int16	0..1024
Battery voltage (physical)	battery	int16	$[10*V]$
Desired thrust front rotor (physical)	forceFront	int16	$[1000*N]$
Setpoint brushless controller front	setpointFront	uint8	0..255
Desired thrust rear rotor (physical)	forceRear	int16	$[1000*N]$
Setpoint brushless controller rear	setpointRear	uint8	0..255
Desired thrust right rotor (physical)	forceRight	int16	$[1000*N]$
Setpoint brushless controller right	setpointRight	uint8	0..255
Desired thrust left rotor (physical)	forceLeft	int16	$[1000*N]$

Setpoint brushless controller left	setpointLeft	uint8	0..255
Total motor force/thrust (physical)	forceTotal	int16	[1000*N]
Remote control connected flag	remoteConnected	bool	0..1
Motors on flag	remoteMotorsOn	bool	0..1
Desired force/thrust (remote)	remoteForceRaw	uint8	0..255
Desired force/thrust (physical)	remoteForce	int16	[1000*N]
Desired yaw velocity (remote)	remoteYawRaw	uint8	0..255
Desired yaw velocity (physical)	remoteYaw	int16	[(1000*rad)/sec]
Desired pitch angle (remote)	remotePitchRaw	uint8	0..255
Desired pitch angle (physical)	remotePitch	int16	[1000*rad]
Desired roll angle (remote)	remoteRollRaw	uint8	0..255
Desired roll angle (physical)	remoteRoll	int16	[1000*rad]

Table 4.4: System states

4.3.3 copter

This is the only module of the Data Layer. It contains the two main data structures of the embedded software. They are defined as module variables. Therefore they can be accessed directly by functions of this module, whereas functions of other modules use

CopterConfig* copterGetConfigPtr(void) and

CopterState* copterGetStatePtr(void)

to get a pointer to the corresponding data structure. Besides these two, this module provides the following functions to the outside:

void copterRestoreConfig(void)

void copterSaveActualConfig(void)

void copterActualizeState(void)

void copterSetRpms(void)

void copterOnBoardDiagnosis(void)

`copterSaveActualConfig()` saves the actual configuration parameters in the EEPROM of the microcontroller and `copterRestoreConfig()` restores them. `copterActualizeState()` actualizes all sensor values within the `copterState` structure by calling all corresponding functions of the HAL. Additionally the raw values of the remote control are converted into physical ones for internal usage without floating point unit – $[1000 \cdot N]$ for thrust, $[1000 \cdot \text{rad/sec}]$ for yaw velocity and $[1000 \cdot \text{rad}]$ for the desired roll and pitch angle. This is done within the `copterActualizeState()` function because the conversion depends on parameters, which are stored in the `copterConfig` structure. The function `copterSetRpm()` sends the setpoints, stored in the `copterState` structure, to the brushless controllers. `copterOnBoardDiagnosis()` checks, if the battery voltage is low (10.5 V) and turns on the beeper, if this is the case. Additionally it the green LED is toggled with each call of the function and the red LED is turned on, if no signal from the remote control is received.

4.4 Application Layer

The Application Layer of the embedded software uses time-triggered tasks only. The most important task is of course the flight control. It comprises three major steps. First, the real-time database is actualized. Second, the control algorithms are executed to calculate the new actuator setpoints and finally these setpoints are sent to the brushless controllers. This procedure runs every ten milliseconds. For telemetry purposes the quadcopter is equipped with a XBee radio module. The software checks every 100 milliseconds if there is a new service request from the base station or if data has to be sent back. In a period of one second, a diagnosis task is scheduled checking the basic parameters and functionalities of the quadcopter. Figure 4.6 shows the program flow in principle. Each task and its sub-functions are implemented in one module.

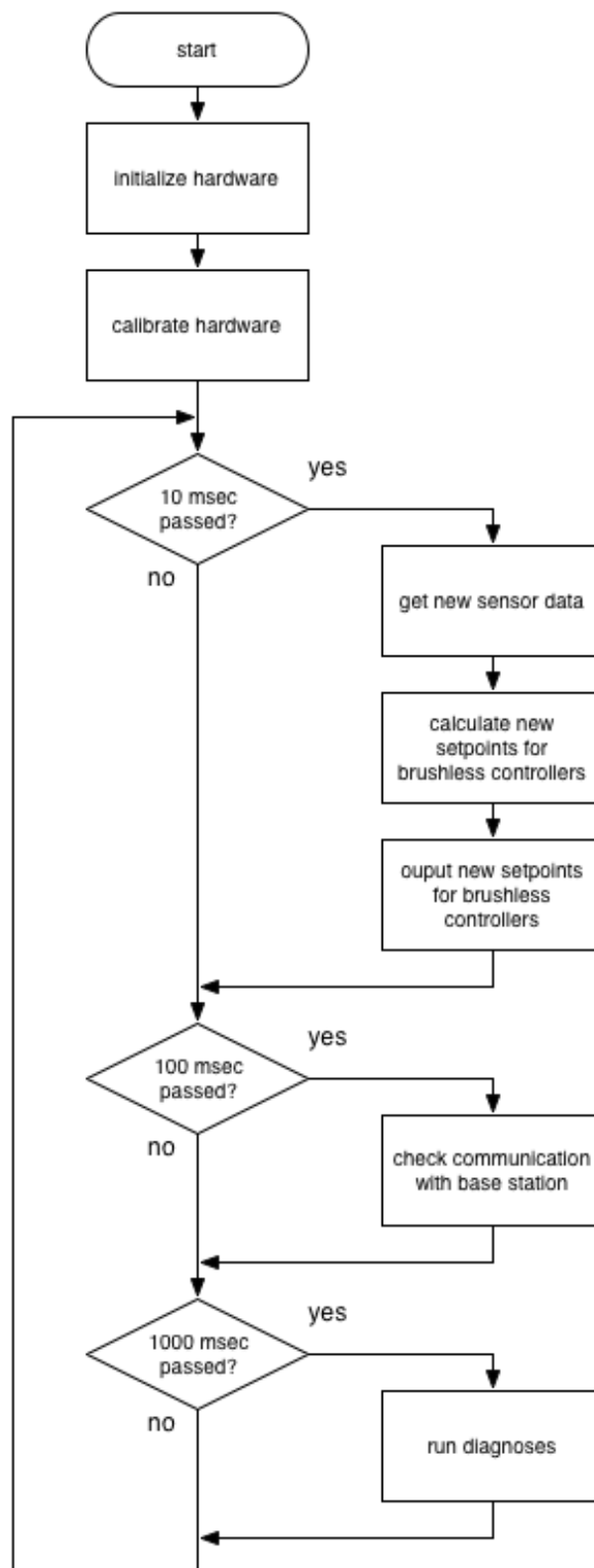


Figure 4.6: Program flow in principle

4.4.1 main

`void main(void)`

This module contains only the `main()` function, the entry point of the embedded software. This function implements the functionality described above using special timer functionality, see also chapter 4.2.9. When the quadrocopter is switched on and the program starts to execute, the hardware is initialized and all interfaces are configured. After one second the sensors are calibrated. Then the program enters an endless loop in which the corresponding tasks are called with the defined period.

4.4.2 basestation

This module implements, together with the `QH_xbee` module, the communication protocol used for communicating with the base station.

It provides the following function to the outside:

`void basestation(void)`

The `basestation()` function checks, if there was a message from the base station received. If this is the case, the message is interpreted and the appropriate response is sent. Additionally *Parameter Response* messages are sent periodically, if this was requested. To fulfill its tasks, the function accesses the central data structures `copterConfig` and `copterState`. It is called every 100 milliseconds.

4.4.3 flightcontrol

In this module the control algorithm of the flight control is implemented.

It provides the following function to the outside:

`void flightControl(void)`

`flightControl()` takes the actual sensor and remote control values and calculates the new setpoints for the brushless controllers. Therefore PID-controllers for roll-angle, pitch-angle and yaw-velocity are used. To read the actual sensor data and to store the calculated setpoints, the `copterState` structure is accessed. The parameters of the controllers are read from `copterConfig`. The `flightControl()` function is executed every 10 milliseconds.

HINT: To improve the behavior of the quadcopter, an additional height controller should be implemented.

4.4.4 sensorfilter

This module is responsible for the calculation of the actual roll- and pitch-angle, which are needed for the roll and pitch controllers.

It provides the following functions to the outside:

`void sensorFilter(void)`

`sensorFilter()` takes the actual acceleration and rotation rate values and calculates the corresponding roll- and pitch-angle. The actual acceleration and rotation rate values are read from, the roll- and pitch-angle are written to the `copterState` structure. The filter coefficients are read from `CopterConfig`.

4.5 Conclusion and further improvements

The embedded software is designed in a modular way. It consists out of three layers – the Hardware Abstraction Layer, the Data Layer and the Application Layer. Therefore new software functions can be added to the Application Layer without changing the underlying software.

The source codes for flight control and sensor filtering are exactly the same as they are used in the Simulink® S-functions (see chapter 2.6). In that way it is ensured that no errors can be made during integration of these software modules. This could only be achieved by using a central type definition file.

During the integration phase there were several problems due to variable overflows. Because no floating-point unit is used, many internal values are multiplied by 100, 1000 or 10000 to get the desired precision e.g. internal representation of angles in [rad]. If such values are multiplied again with each other, the value range of an `int16` variable can be exceeded easily. This is one point, which must be regarded in future as well.

To further improve the embedded software, the following points can be taken into consideration:

- Implementation of a height control by using the air pressure sensor in combination with the other sensors
- Implementation of an emergency landing mode, which can be used e.g. when the connection to the remote control is lost (until now, the motors are switched off)
- Use of the EEPROM to save the actual quadrocopter implementation (already implemented but commented out because reprogramming of the microcontroller clears the EEPROM → wrong parameters would be loaded on startup)
- Reading back values from the brushless controllers to improve the setpoint calculation (unfortunately there is less documentation available regarding this functionality; please see the board of www.mikrokoetter.de)
- Use of the self-test functionality of the accelerometer and the gyroscopes to ensure that the sensors work fine (maybe also to improve the sensor calibration process)
- Use of the temperature sensor in the gyroscope to compensate temperature drift
- Use of the data ready signal of the accelerometer, which currently causes infinite waiting times

5 Quadrocopter – Base Station Communication

The data, transferred from and to the base station, is used for visualizing the attitude and actual state of the system as well as for changing system parameters. XBee Pro RF modules are used to transfer the data wireless.

5.1 General Information

For the communication between quadrocopter and base station there is an own protocol used for synchronization and integrity. This is necessary because an UART interface is byte oriented and there are much longer messages to be transmitted. The protocol has the following properties:

- The base station initiates all communication.
- For transmission error detection and synchronization purposes, a CRC16 checksum is added to each message (generator polynomial: 0xA001).
- The MSB (Most Significant Bit) of a byte/value is always sent first. This holds also for 16-bit values (Bits: [16...8] [7...0]).
- A message consists out of identifier, data length field, data field and checksum. The maximum message length is 256 byte (252 byte payload). For details, please see chapter 5.2.
- There must be a gap of at least 200ms between two consecutive messages sent from the base station. Otherwise messages are lost in the flight control because of a buffer overflow/overwrite. The gap between two bytes of a message must not be larger than 2ms.

5.2 Basic message layout

First, an one byte ID is transmitted specifying the message type. The second byte represents the number of payload bytes (Data Length Code – DLC). The payload itself can vary between zero and 252 bytes. A 16-bit checksum (Cyclic Redundancy Check – CRC) finishes the message.

The basic layout is illustrated in Figure 5.1. Each block represents one byte. The content of the ID and data field is described for each message separately. The CRC is calculated over the ID, DLC and the data field.

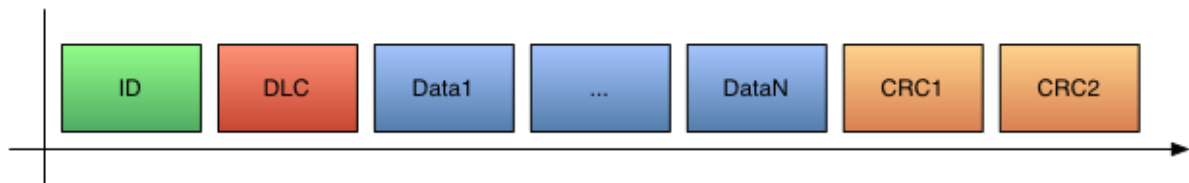


Figure 5.1: General message layout

5.3 Physical data transmission

For communication with the XBee modules over UART (RS232) the following configuration is used:

- 9600 baud/sec
- 1 start bit, 8 data bits, 1 stop bit
- No parity

All messages, which are sent between quadrocopter and base station, are defined on the next pages. Other messages will be ignored.

CAUTION: In general it must be ensured that the communication between quadrocopter and base station has no bad influence on the flight control task. E.g. the transmission of a message from the quadrocopter to the base station blocks the flight control task longer than its calculation period.

CAUTION: Because the wireless link between the XBee modules is a half-duplex communication, it must be ensured that both sides are able to transmit data. Neither the quadrocopter nor the base station is allowed to occupy the wireless channel completely by sending data all the time.

5.4 Parameter Request (base station → quadrocopter)

This message is sent from the base station in order to request the actual calibration parameters of the quadrocopter (see chapter 4.3.1). When the quadrocopter receives this message, it has to respond with a *Parameter Response* message.

ID: 0x10

DLC: 0x00 (0)

Data: None

5.5 Parameter Response (quadrocopter → base station)

This message is sent from the quadrocopter, when a *Parameter Request* or a *Parameter Update* message was received. The data field contains the actual calibration parameters in the same order as Table 4.3 shows.

ID: 0x11

DLC: 2A (42)

Data: All calibration parameters according to Table 4.3

5.6 Parameter Update (base station → quadrocopter)

This message is sent from the base station in order to update the calibration parameters of the quadrocopter. The message will only have an effect, if the quadrocopter is at the ground (RPM of all motors equal to zero). The data field contains the new parameters in the same order as in Table 4.3. When the quadrocopter receives this message, it has to respond with a *Parameter Response* message. When it is up in the air, the actual parameters are sent back.

ID: 0x12

DLC: 2A (42)

Data: All calibration parameters according to Table 4.3

5.7 Status Request (base station → quadcopter)

This message is sent from the base station in order to request the actual system status (see chapter 4.3.2). Dependent on the data byte, the quadcopter either responds only once or sends periodically a *Status Response* message. A data byte value of 0xFF requests a single response message. For all other values, the data byte specifies the response period. Response period = data byte value * 100 msec (execution period of the `basestation()` function). Sending 0x00 in the data field stops a periodical response of the quadcopter.

ID: 0x20

DLC: 0x01 (1)

Data: One byte: 0x00 – stop periodical response

0x01 – *Status Response* message every 100 msec

0x02 – *Status Response* message every 200 msec

0x05 – *Status Response* message every 500 msec

0x0A – *Status Response* message every 1000 msec

0xFF – single *Status Response* message

There are only some examples for the data byte value listed above. All other values in between are also possible.

5.8 Status Response (quadcopter → base station)

This message is sent from the quadcopter either once or periodically dependent on the received *Status Request* message. The data field contains the actual status values in the same order as in Table 4.4.

ID: 0x21

DLC: 0x42 (66)

Data: All actual status/sensor values; for ordering see Table 4.4.

6 Basestation Software

In this section consists information about basestation software. It is divided on three sub chapters, which are describes

- installation of relevant drivers
- structure of designed software
- user interface guide

6.1 Installation guide

6.1.1 Installation Com port drivers

There is a trick to install the Java Communications API correctly on a Windows system Machine. The following files are the core of JAVA Communication API, and they are very important to have them installed on your system for a proper operation:

- comm.jar
- win32com.dll
- javax.comm.properties

For the JDK (Java Development Kit) to recognize the serial ports on your machine, it is important to properly place these files in the right folders on your local machine: %Java_HOME% = the location of your JDK directory.

To find your JDK directory, Use the Following steps:

1. Click on "Start"
2. Click on "Search"
3. Click on "For Files or Folders ..."
4. In the left hand side, click on "All Files and Folders"
5. Type in jdk* in the textbox under all or part of the file name
6. Click "Search"
7. Look for yellow icon that looks like a folder
8. Double click on the folder to open the JDK folder

comm.jar should be placed in:

- %JAVA_HOME%/lib
- %JAVA_HOME%/jre/lib/ext

win32com.dll should be placed in:

- %JAVA_HOME%/bin
- %JAVA_HOME%/jre/bin
- %windir%System32

javax.comm.properties should be placed in:

- %JAVA_HOME%/lib
- %JAVA_HOME%/jre/lib

6.1.2 Installation of libraries for JFreeChart

Overview

NetBeans (<http://www.netbeans.org/>) is a free IDE developed by Sun Microsystems. In NetBeans, third party libraries are configured using the Library Manager.

I'll describe how to set up JFreeChart and JCommon within the Library Manager in NetBeans.

Version 5.5. This makes it straightforward to include JFreeChart and JCommon as dependencies in your application(s), with NetBeans automatically handling features like code completion, Javadoc popups, stepping through the JFreeChart/JCommon sources during debugging, and more.

Configuration Steps

To begin with, you need to download the JFreeChart and JCommon distributions, unpack them on your local machine, and generate the API documentation.

The following steps are necessary:

1. Download the latest version of the JCommon class library: <http://www.jfree.org/jcommon/> and unpack it to a directory on your computer (almost anywhere is fine).
2. From the ant subdirectory of the just-unpacked JCommon, run ant javadoc to generate the Javadocs locally. If you are unfamiliar with Ant, you can skip this step, but then NetBeans won't be able to show you the Javadoc popups for JCommon.
3. Download the latest version of the JFreeChart class library: <http://www.jfree.org/jfreechart/> and unpack it to a directory on your computer (again, almost anywhere is fine).
4. From the ant subdirectory of the just-unpacked JFreeChart, run ant javadoc to generate the Javadocs locally. As with step 2, you can skip this step, but then you'll be missing the API documentation. Now, launch NetBeans, and carry out the following steps to configure JFreeChart and JCommon as user libraries:
5. In NetBeans, select the Library Manager item from the Tools menu; you should see the dialog shown in Figure 6.1.

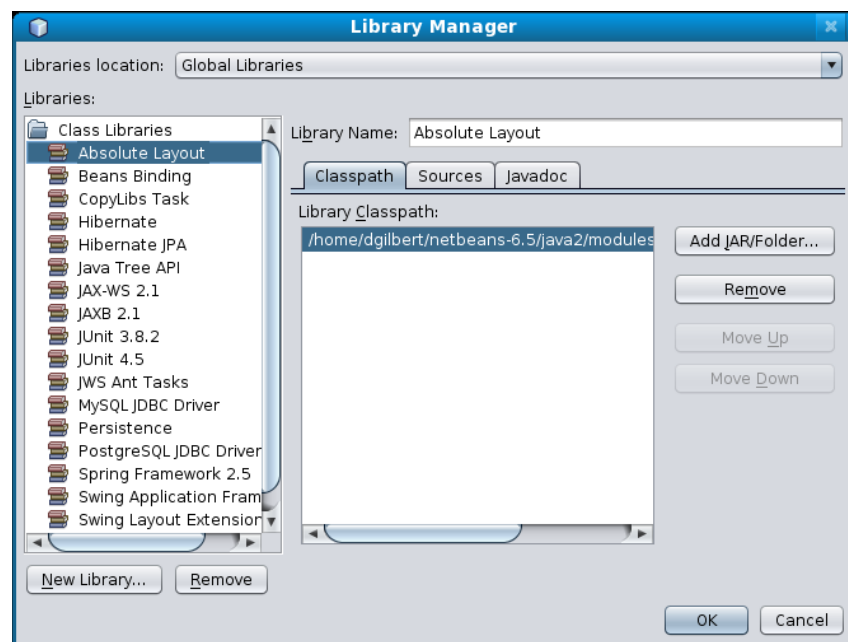


Figure 6.1: The Library Manager

6. Click on the "New Library..." button and enter JCommon-1.0.16 as the library name.
7. With the "Classpath" tab selected, click on the "Add JAR/Folder..." button and select the jcommon-1.0.16.jar file from the JCommon directory created back in step 1.
8. With the "Sources" tab selected, click on the "Add JAR/Folder..." button and select the source directory for JCommon.
9. With the "Javadoc" tab selected, click on the "Add ZIP/Folder..." button and select the javadoc directory for JCommon (refer to step 2).
10. Click on the "New Library..." button and enter JFreeChart-1.0.13 as the library name.
11. With the "Classpath" tab selected, click on the "Add JAR/Folder..." button and select the jfreechart-1.0.13.jar file from the JFreeChart directory created back in step 3.
12. With the "Sources" tab selected, click on the "Add JAR/Folder..." button and select the source directory for JFreeChart.
13. With the "Javadoc" tab selected, click on the "Add ZIP/Folder..." button and select the javadoc directory for JFreeChart (refer to step 4).

At this point, you have completed the configuration of the libraries. The next section shows how to create a new project in NetBeans that depends on these libraries.

6.1.3 Installation X-CTU Software & USB drivers

Use installation disk of X-CTU. Documentation is attached.

6.2 Software description

The main class in the SW Helicopter is Display. It creates main Frame and connects the main classes of the program. Figure 6.2 shows interconnections between classes. Instance of this class has the JFrame type.

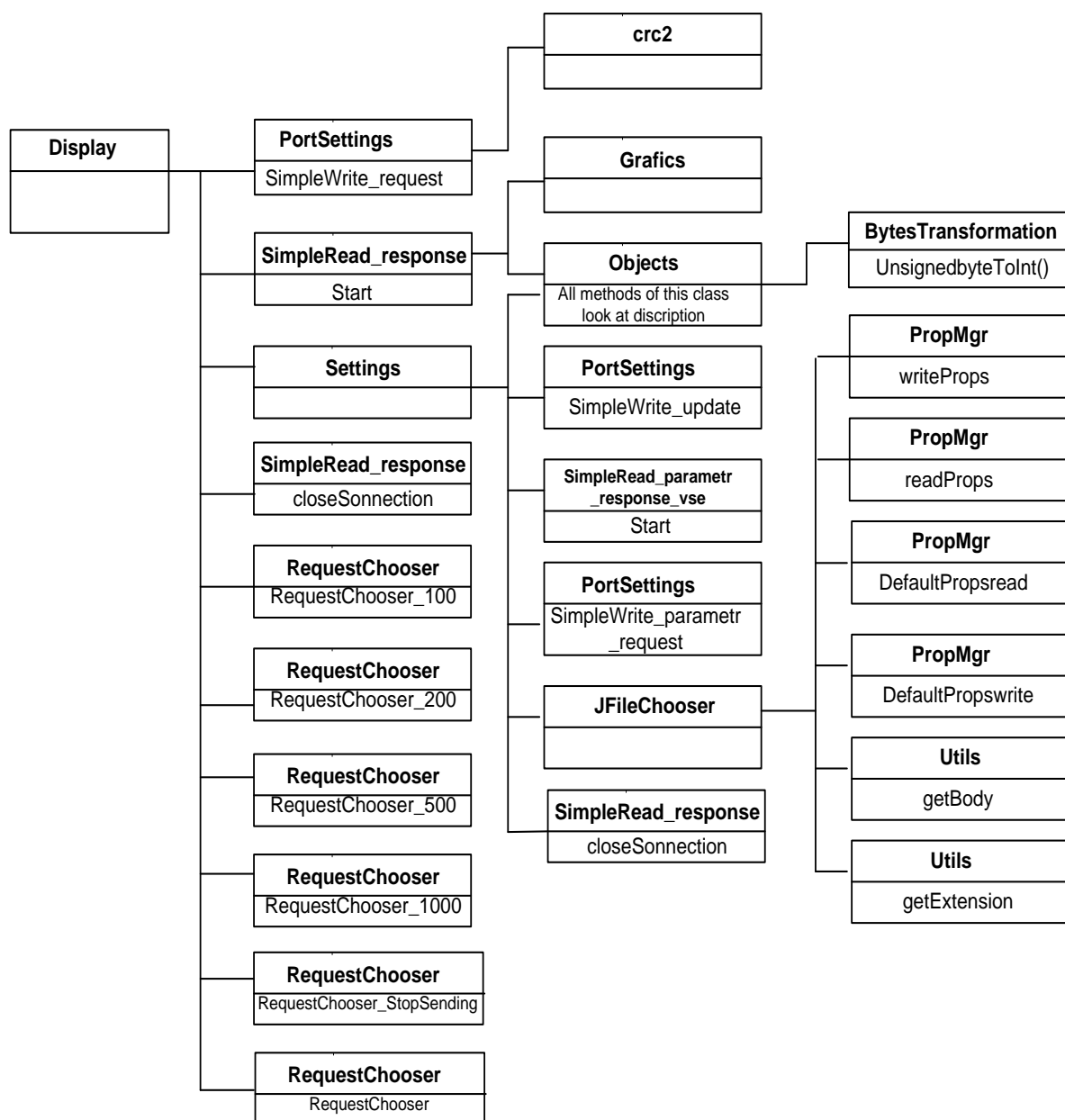


Figure 6.2: Class interconnection tree

6.2.1 class PortSettings

One of the main classes of this program. It allows user to open com-port and send the request user need to helicopter. Automatic detection of com-port existence is implemented in FindPortIdentification method. It makes it easy to detect and open com-port if it is exist. The following settings of com-port were used:

- 9600
- SerialPort.DATABITS_8
- SerialPort.STOPBITS_1
- SerialPort.PARITY_NONE

These parameters can be changed in the future if it is necessary.

In all methods output and input data streams was used DataOutputStream and DataInputStream. Standard method to send primitive data to the stream was implemented:

- writeByte
- writeChar
- writeShort
- and so on

In addition this class content method for testing work (sending/receiving data to/from helicopter/file).The name of the method is SimpleWrite. Working with the file on main station it taking data from preformed file "peredacha.data" and send it to the helicopter via XBee as a stream of primitive data.

Variables

- public static CommPortIdentifier thePortID
- public static CommPort thePort

Methods

- `public static void simplewrite_request(byte Request)`
 - forming the message to request parameters from the helicopter
 - calculating the CRC
 - open and initialisation of com-port
 - sending the message to helicopter
- `public static void simplewrite_update()`
 - getting the parameters of desirable values of control parameters from the textfield of the Frame
 - calculating the CRC
 - open and initialisation of comport
 - sending the message to helicopter
- `public static void simplewrite_Parametr_request()`
 - sending the request for getting the parameters from helicopter
 - calculating the CRC
 - open and initialisation of comport
 - sending the message to helicopter
- `public static void simplewrite_response()`
 - was created to check the functionality of this program by connection two computer via XBee
 - sending the test message with predefined values
- `public static String FindPortIdentification()`
 - automatic identification of com-port
- `public static void simplewrite()`
 - was created for testing work of sending/receiving data to/from helicopter/file.

6.2.2 class BytesTransformation

This class content methods which make all transformation with primitive variables:

- unsigned byte to integer
- char to two bytes array

There no unsigned data in Java available.

Type of data	Range	
byte	-128	127
short	-32,768	32,767
int	-2,147,483,648	2,147,483,647

Table 6.1: Java primitive's data types

Because Java treats the byte as signed, if its unsigned value is above >127, the sign bit will be set (strictly speaking it is not "the sign bit" since numbers are encoded in two's complement) and it will appear to java to be negative.

The two's complement of a binary number is defined as the value obtained by subtracting the number from a large power of two (specifically, from 2^N for an N-bit two's complement). The two's complement of the number then behaves like the negative of the original number in most arithmetic, and it can coexist with positive numbers in a natural way.

most-significant bit										
0	1	1	1	1	1	1	1	=		127
0	1	1	1	1	1	1	0	=		126
0	0	0	0	0	0	1	0	=		2
0	0	0	0	0	0	0	1	=		1
0	0	0	0	0	0	0	0	=		0
1	1	1	1	1	1	1	1	=		-1
1	1	1	1	1	1	1	0	=		-2
1	0	0	0	0	0	0	1	=		-127
1	0	0	0	0	0	0	0	=		-128

Table 6.2: The two's complement

As the project requires of using unsigned short data type instead of short data type char was implemented. For transforming unsigned byte to integer value UnsignedbyteToInt method was written.

Methods

- public static int UnsignedbyteToInt(byte first)
 - transforming unsignedbyte to integer
- static byte[] charToByteArray(char c)
 - transforming char to two bytes array

6.2.3 public class Objects

Was created to organize transparent structure of incoming and out coming data, reconstructing the message sent by helicopter. This class uses BytesTransformation class for transformation of primitive data.

6.2.4 public class SimpleRead_response

Implements Runnable, SerialPortEventListener.

This class allows to initialize and open com port for reading information from the helicopter via XBee. As it is implement Serial Port Event Listener the methods of this class is available only if some data come to the com port. This method revokes the chart for reflecting the data from helicopter. This chart is dynamic and data are changing as fast as it is coming to the serial port. In addition the method constructs the messages. Contents of the message is formed from the message which has come to the Serial Port Event Listener. What to include in displayed message depends on user desire. It might be changed and adopted easily if user needs so. To added the new parameter standard method text.append was used. For termination of connection closeSonnection() mas written. It removes Com Port Listener and close the Serial Port.

Variables

- `Grafics graf`
- `static CommPortIdentifier portId`
- `static Enumeration portList`
- `InputStream source`
- `InputStream inputStream`
- `SerialPort serialPort`
- `Thread readThread`
- `ByteArrayInputStream bais`
- `JTextArea text = new JTextArea(10,10)`

Methods

- `public static void start()`
 - inisialisation
 - opening the com-port
- `public SimpleRead_response()`
 - retrieves the graphical part of the program
 - getting information from the helicopter
 - display it in `TextArea` as a text
- `public static void closeSonnection()`

6.2.5 `public class Grafics`

Extends `ApplicationFrame`, implements `ActionListener` `Grafics`.

This method was written to implement dynamic Charts for reflecting the data coming on Serial Com Port. The core of this class is using of `TimeSeriesCollections`. The Method draws 4 dynamic charts. Data for the set of variables are coming from the port. As it is dynamic the set is always filled in. At this Application the data for the charts is the Force of the forth rotors. If user wants it is possible to extend the chart, make it filled by multiple curves of add some extra charts. This method is very flexible for using and future upgrades.

Variables

- `public static final int SUBPLOT_COUNT = 4`
- `private TimeSeriesCollection[] datasets`
- `private double[] lastValue = new double[SUBPLOT_COUNT]`

Methods

- `public Grafics(final String title)`
 - creates a graphics. The number of graphics is `SUBPLOT_COUNT`

6.2.6 public class Setting

Extends javax.swing.JFrame.

This is one of the main class. First of all it creates the form for filling by operator. It helps user to work with data:

- saving
- getting
- reading parameters of helicopter

This class uses PropMgr, Utils, PortSettings, SimpleRead_parametr_response_vse classes.

6.2.7 public class SimpleRead_parametr_response_vse

Implements Runnable, SerialPortEventListener

This class allows to initialize and open com port for reading information from the helicopter via XBee. As it is implement Serial Port Event Listener the methods of this class is available only if some data come to the com port. SimpleRead_parametr_response_vse constructs a new Frame for reflecting the data in TextField. This is the static data which described the setting of the helicopter:Control, desiable parameters, filters and so on.

Variables

- Grafics demo
- static CommPortIdentifier portId
- static Enumeration portList
- static String t
- String FILENAME = "binary.dat"
- InputStream source
- InputStream inputStream
- SerialPort serialPort
- Thread readThread
- ByteArrayInputStream bais
- JEditorPane text1 = new JEditorPane()

- `JEditorPane text2 = new JEditorPane()`
- `static char v[] = new char [17]`
- `TextField kpR=new JTextField()`
- `JTextField kiR=new JTextField()`
- `JTextField kdR=new JTextField()`
- `JTextField kpP=new JTextField()`
- `JTextField kiP=new JTextField()`
- `JTextField kdP=new JTextField()`
- `JTextField kpY=new JTextField()`
- `JTextField kiY=new JTextField()`
- `JTextField kdY=new JTextField()`
- `JTextField lpR=new JTextField()`
- `JTextField hpR=new JTextField()`
- `JTextField lpP=new JTextField()`
- `JTextField hpP=new JTextField()`
- `JTextField inertiaX=new JTextField()`
- `JTextField inertiaY=new JTextField()`
- `JTextField inertiaZ=new JTextField()`
- `JTextField lenBoom=new JTextField()`
- `JTextField forceMax=new JTextField()`
- `JTextField angVelYMax=new JTextField()`
- `JTextField angPMax=new JTextField()`
- `JTextField angRMax=new JTextField()`

Methods

- `public static void start()`
 - initialisation of com port
- `public SimpleRead_parametr_response()`
 - getting static data from the helicopter and create the Frame, where it displays them

6.2.8 public class Utils

“Utils” was created as help for Chooser. As this class can find the boarder between the title of file and its extension, chooser uses it in its methods for saving or opening the users setting or downloading the default settings.

Methods

- public static String getExtension(File f)
 - getting extension from the chosen file name
- public static String getbody(File f)
 - getting the body of the chosen file name

6.2.9 public class PropMgr

This class was created to work with properties file. It can

- write default file with properties
- read the default properties
- read properties from the chosen properties file
- write properties to the desirable file

For working with class following standard methods were implemented: setProperty, getProperty. Working with input/outputStream.

Methods

- public static void defaultPropswrite()
 - write default properties to the file "defaultProps2"
- public static String defaultPropsread(String t)
 - read the default properties to the file "defaultProps2"
- public static String readProps(String otkuda,String t)
 - read properties from the chosen properties file
- public static void writeProps
 - write properties to the desirable file

6.2.10 public class crc2

Calculates the cyclic redundancy check. CRC is calculated for each sending or receiving messages which are incoming or out coming to/from the Com port via XBee. For calculation cyclic redundancy check 0xA001 generator polynomial is used.

Variables

- int crc
- int buffer
- byte i
- byte j

Method

- public int crc2 byte(byte[] data,int n)

6.2.11 public class readfromfile_mod()

Class for testing procedure; reading from the file.

Variables

- public static DataInputStream inDataStream
- public static int[] v = new int[30]
- long de
- static int i = 0

Method

- public int[] readfromfile_mod()
 - read from the file put the data to the outcoming stream

6.2.12 public class writetofile

Class for testing procedure; writing from the file

Method

- public writetofile()
 - writing some numbers to the file

6.2.13 public class RequestChooser()

This class was developed to fulfill the necessity of sending request to the helicopter. This request consisted information of desirable time of dynamical answer of helicopter. As described in chapter 4 answer from helicopter should be by request every:

- 100ms
- 200ms
- 500ms
- 1000ms

or it might be just single answer. That is why following method were created:

Methods

- public static byte RequestChoooser()
 - Sending request which implies only answer by single message. Byte for recognition request is 128.
- public static byte RequestChoooser_100()
 - Sending request which implies answers every 100ms. Byte for recognition request is 0x1.
- public static byte RequestChoooser_200()
 - Sending request which implies only answer every 200ms. Byte for recognition request is 0x2.

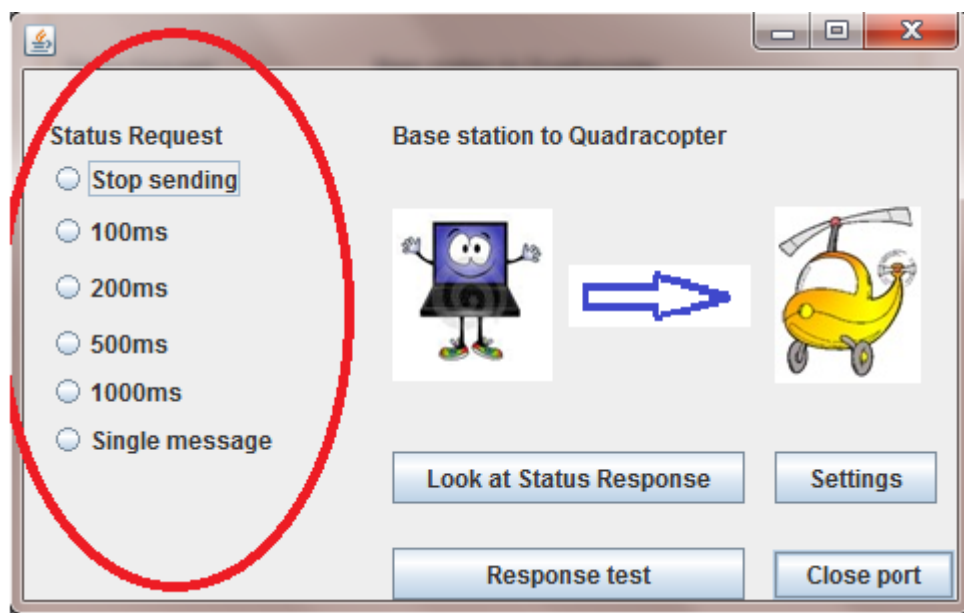
- `public static byte RequestChoooser_500()`
 - Sending request which implies only answer every 500ms. Byte for recognition request is 0x5.
- `public static byte RequestChoooser_1000()`
 - Sending request which implies only answer every 1000ms. Byte for recognition request is 0xA.
- `public static byte RequestChoooser_StopSending()`
 - Sending request which asks helicopter not to send any periodical data. Byte for recognition request is 0x00.

6.3 User Interface

See also chapter 5.

6.3.1 Status Request

The main window of program fulfills the requirements to make a status request with possibilities to choose period interval (requirement 5.7).



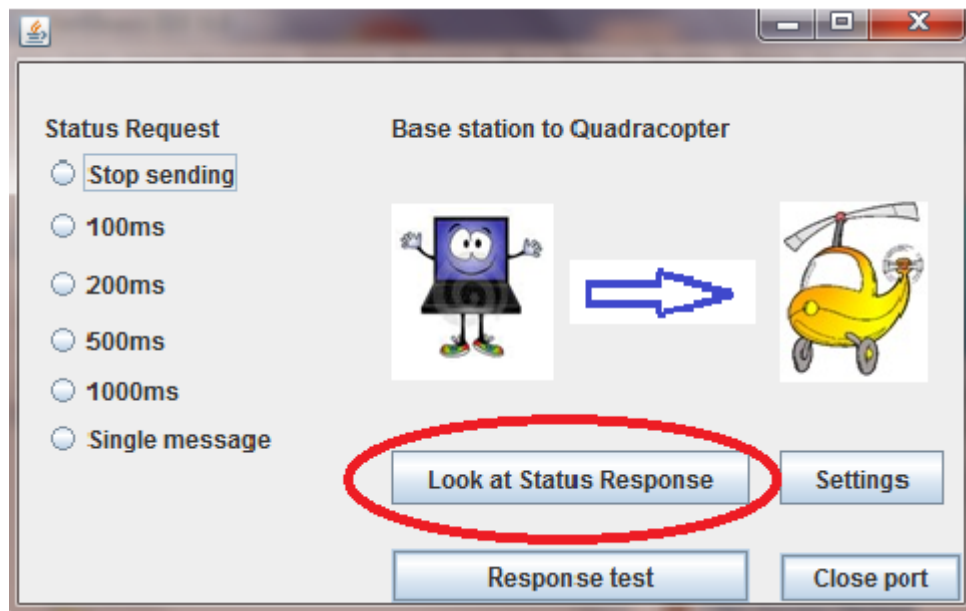
Program detects the com-port automatically and try to send request.

The message consists information about ID, DLC, chosen request byte and CRC

0x20	0x01	request	CRC
------	------	---------	-----

The CRC is calculated by BytesTransformation class. Method CRC This class is used always when it is necessary to make any manipulation with bytes.

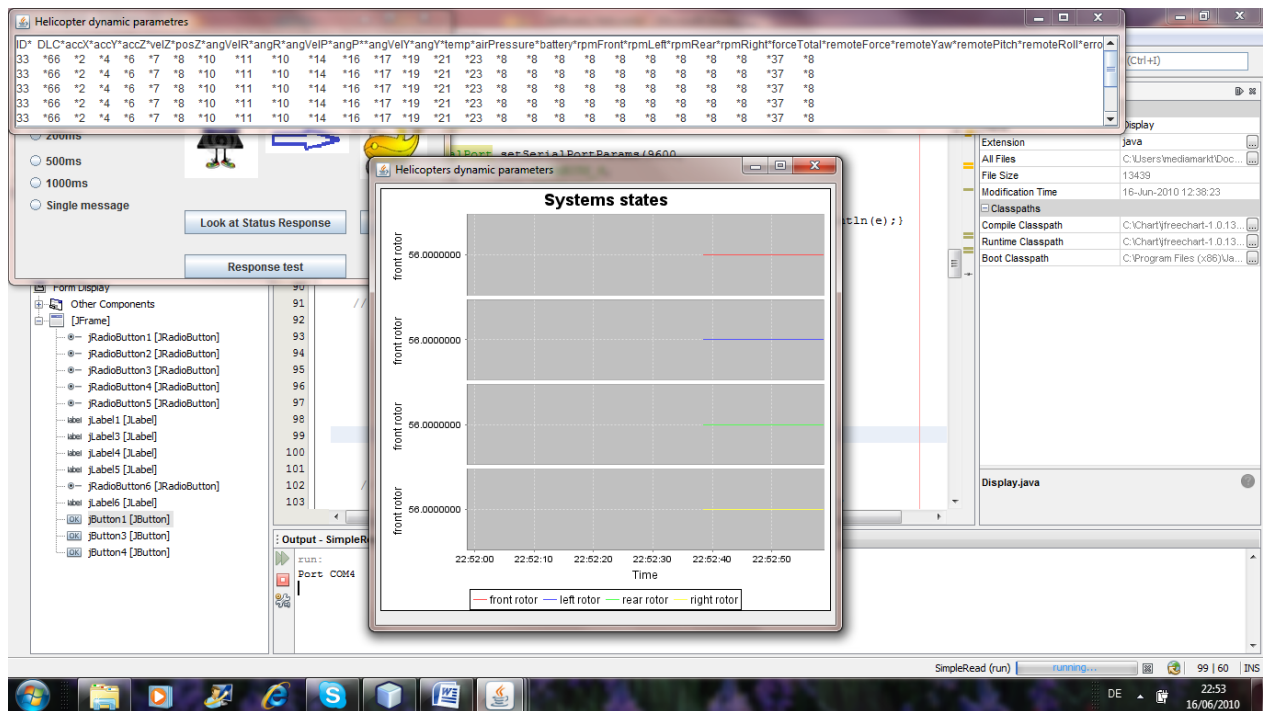
6.3.2 Status Response



Pressing "Look at Status Response" calls `SimpleRead_response.Start()`.

This class obtains the data from the com-port. In this class we predefined the order of coming message regarding Table 4.4.

As we have a lot of coming parameters only a few are reflected in the GUI. Speed: front rotor, left rotor, rear rotor, right rotor. All other parameters except raw data are include in printed variant of message.



6.3.3 Parameter request

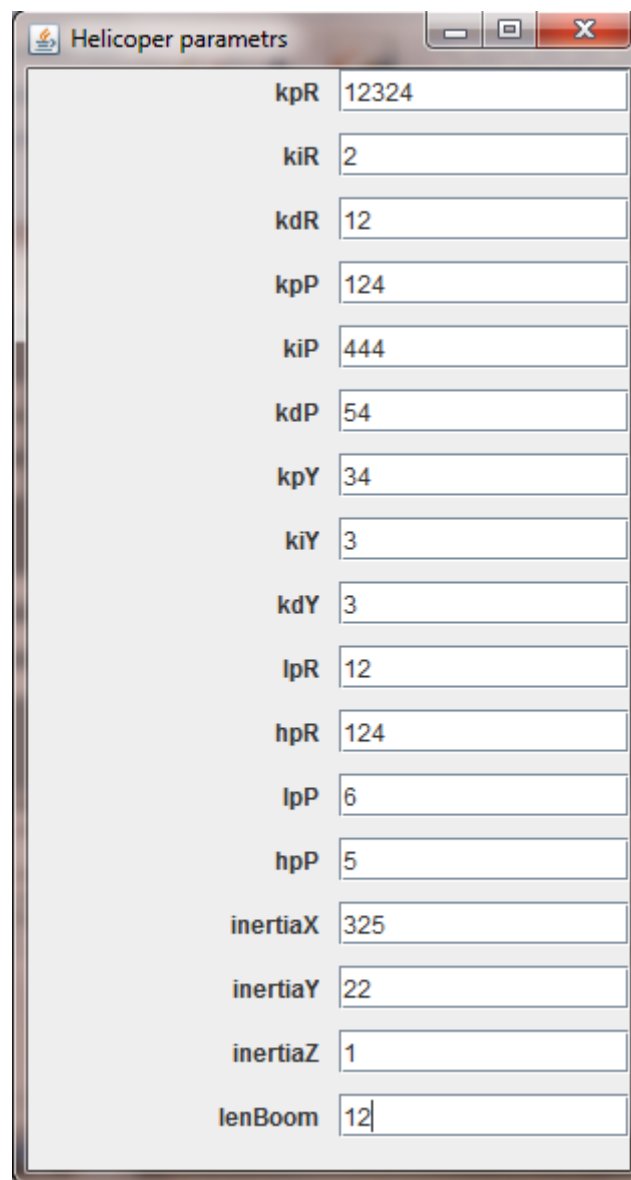
Sending parameter request from base station to helicopter possible with use of page "Settings".

6.3.4 Parameter Update

Taking parameters from text areas of this form and forming the message.

6.3.5 Parameter Response

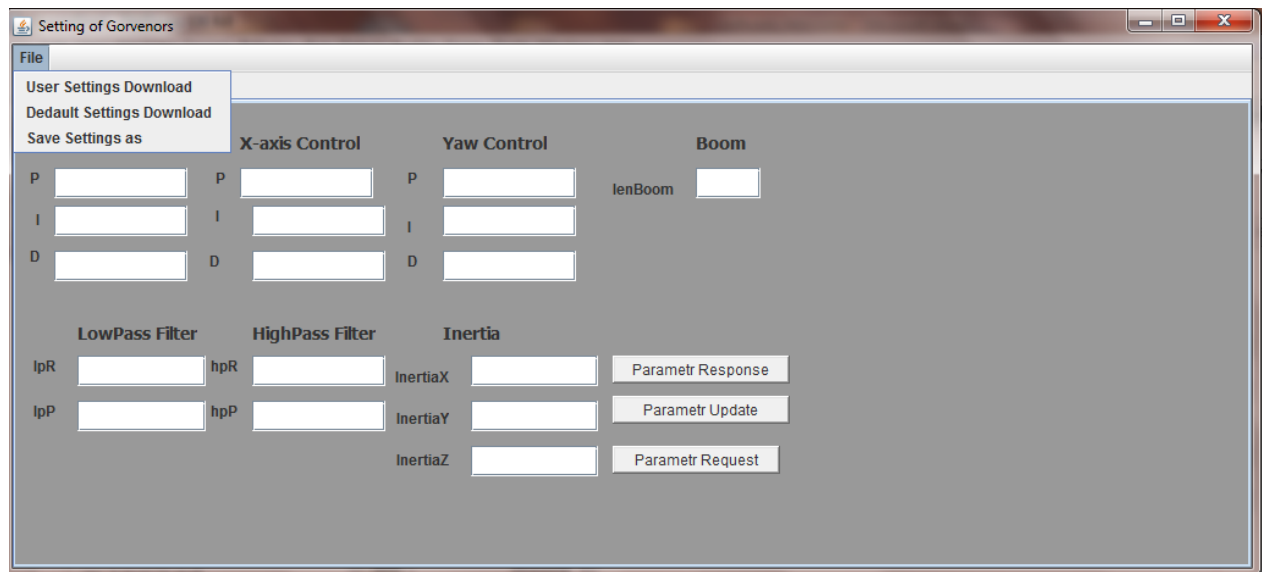
Displays static parameters of helicopter which are send via wireless communication.



Parameter	Value
kpR	12324
kiR	2
kdR	12
kpP	124
kiP	444
kdP	54
kpY	34
kiY	3
kdY	3
lpR	12
hpR	124
lpP	6
hpP	5
inertiaX	325
inertiaY	22
inertiaZ	1
lenBoom	12

6.3.6 Chooser

From the Settings of Governors it is possible to save settings or download default settings.



6.4 Summary

To sum up all work, which was made in the base station software section the following points can be mentioned:

- Developed software fulfilled the functional requirements which were described in chapter 5
- It forms the sending message according to the predefined protocol (look at chapter 5)
- Human-machine interface which is implemented in this software was developed as supplementary part of the project
- Program can be used as significant help during the testing procedure. For this reason the dynamic graphics and messages were displayed. It shows, from one hand, dynamic data of rapidly changing parameters, to another hand –static parameters of controls systems, which can be adjusted.
- Using chooser makes program useful for saving user's setting or download default.

A lot of work was done but there are some drawbacks, which can be enhanced in the future:

- As system is very sensitive to the installed operation systems, user has to fulfill all requirements concerning demands of the system or if it is impossible to do, user has to contact the support service of XBee to get proper drivers.
- The last testing of the program shows that there is a bug in the automatic com-port detection. For future implementation it is necessary to implement the chooser of com with list of available com ports on the computer. Only then user chooses to which com-port from the list of available port you have to connect.

The main conclusion is that this program might be considered as a very good base for the testing and data visualization but after mending up mentioned drawbacks.

7 Conclusion

The overall goal of this project was to have a stable flying quadrocopter. The reference was the mikrokopter project (www.mikrokopter.de). Additionally we got a new central control unit, developed by students of the faculty of mechatronics of the University of Applied Sciences Esslingen. The Frame and all other parts are commercially available.

The outcome of this project is a simulation model of the quadrocopter and the control algorithms as well as the embedded software. The simulation models were developed in MATLAB/Simulink®. The control algorithms are implemented in the simulation as Simulink® S-functions and the source code can be used within the embedded software of the quadrocopter directly. With the developed base station software it is possible to change the system parameters without reprogramming the central control unit and to watch the actual values of the system states. The physical model, the control algorithms, the embedded software and the base station software were individually tested in the first step followed by the integration of all modules. Afterwards the complete system was tested by using two different test benches. In one test bench, one axis of the quadrocopter was fixed via two sliding bearings. It was used to develop and calibrate the roll and pitch control. A second test bench, in which the quadrocopter was tethered vertically, was used to develop and calibrate the yaw velocity control.






The response on the test benches is satisfactory. The quadrocopter accepts all commands and reacts in the desired way. In addition, the dynamic response is okay. It reached the desired states without big oscillations.

Unfortunately it doesn't show the same behavior outside of the test benches. Nearly before liftoff, the quadrocopter tends to flip in a random direction, making it uncontrollable. This problem must be investigated in detail in a future project.

Suggestions for further improvements can be found in the conclusions of chapter 1, 2, 4 and 6.

8 Appendix

8.1 Weight of Frame Parts

Base plate 23g	
Middle plate 13g	
Upper plate 12g	
Fixings 10g	
Flight control 52g	







Brushless control 17g	
Arm 10g	
Fixing 1g	
Motor carrier 5g	
Motor 66g	
Propeller 7g	

Table 8.1: Weight of quadrocopter parts

8.2 Schematic of the central control unit

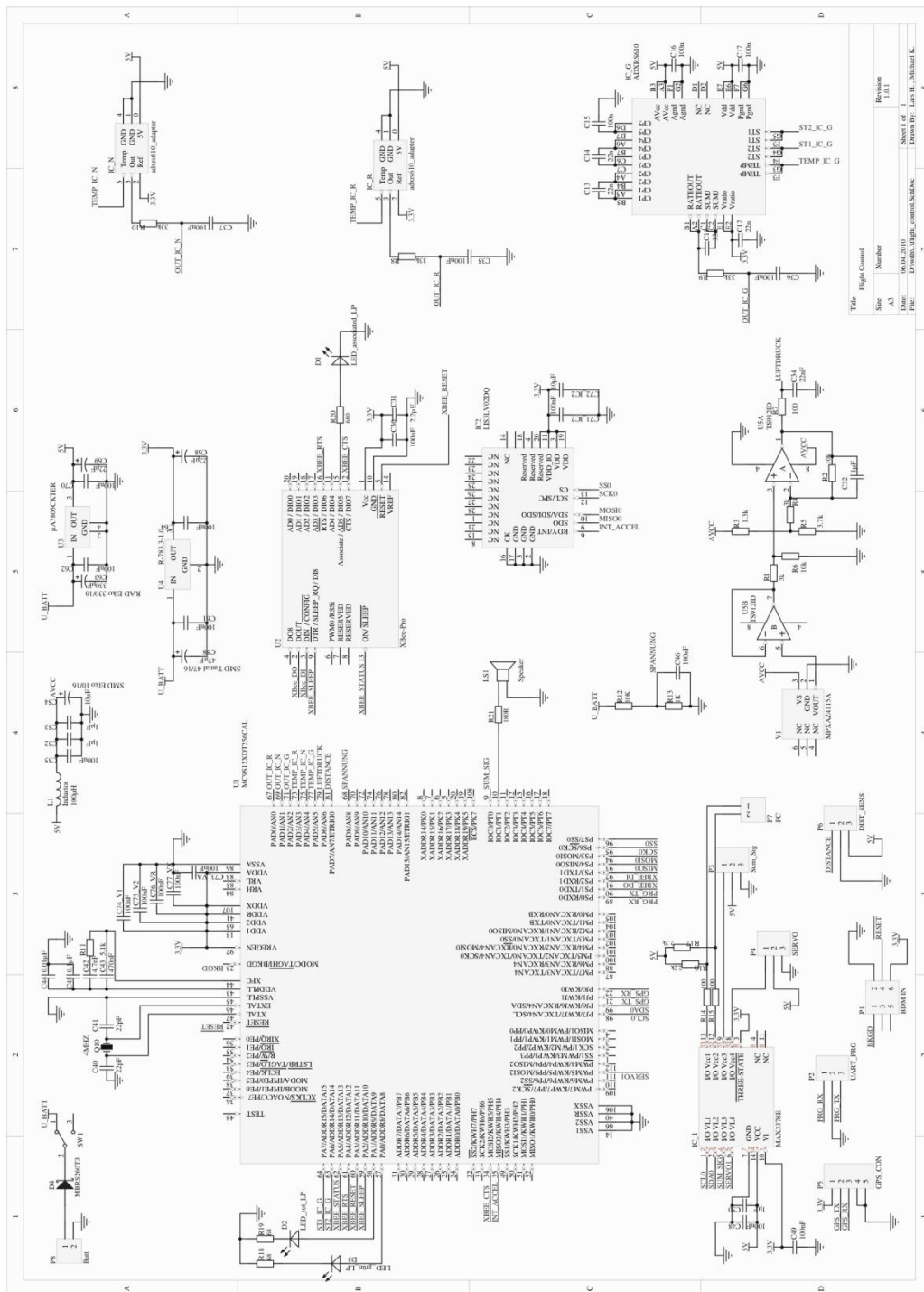


Figure 8.1: Schematic of the central control unit

8.3 Subsystem Interfaces

This section gives an overview of the various interfaces (like signal, type, resolution, unit) between the subsystems of copter system as implemented in the MATLAB/Simulink®.

As mentioned in the Section 2.6, following subsystems are implemented in Simulink®'s S-Function (C-code).

- Quadcopter Controller
- Sensor Filter

8.3.1 Subsystem Quadcopter Controller

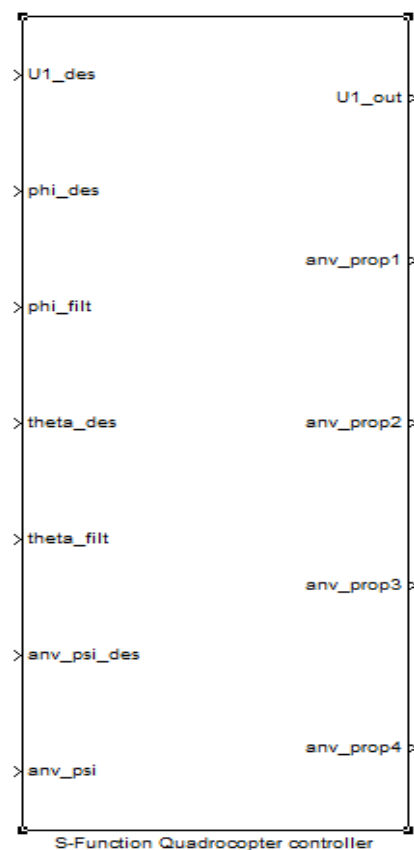


Figure 8.2: Subsystem Quadcopter Controller

Inputs

Signal	Type	Resolution	Unit
U1_des	int16	0.001	N
phi_des	int16	0.001	rad
phi_filt	int16	0.001	rad
theta_des	int16	0.001	rad
theta_filt	int16	0.001	rad
anv_psi_des	int16	0.001	rad/s
anv_psi	int16	0.001	rad/s

Table 8.2: Inputs for Subsystem Quadrocopter Controller

Outputs

Signal	Type	Resolution	Unit
U1_out	int16	0.001	N
anv_prop1	int16	1	rpm
anv_prop2	int16	1	rpm
anv_prop3	int16	1	rpm
anv_prop4	int16	1	rpm

Table 8.3: Outputs for Subsystem Quadrocopter Controller

Parameters

Signal	Type	Resolution	Unit	Default Value
Kp_phi	int16	0.01	1/s ²	20.00
Ki_phi	int16	0.01	1/s ³	5.00
Kd_phi	int16	0.01	1/s	3.00
Kp_theta	int16	0.01	1/s ²	20.00
Ki_theta	int16	0.01	1/s ³	5.00
Kd_theta	int16	0.01	1/s	3.00
Kp_anv_psi	int16	0.01	1/s	20
Ki_anv_psi	int16	0.01	1/s ²	2
Kd_anv_psi	int16	0.01	-	2
I_x	int16	0.001	Nms ²	0.009
I_y	int16	0.001	Nms ²	0.009
I_z	int16	0.001	Nms ²	0.016
I_boom	int16	0.001	m	0.166

Table 8.4: Parameters for Subsystem Quadcopter Controller

Constants

Signal	Value
S_FUNCTION_NAME	sfun_Copter_Controller_discrete
S_FUNCTION_LEVEL	2
NUMINPUTS	7
NUMOUTPUTS	5
NUMPARAMS	1
THRUST2RPM_DATAPOINTS	10

Table 8.5: Constants for Subsystem Quadcopter Controller

8.3.2 Subsystem Sensor Filter

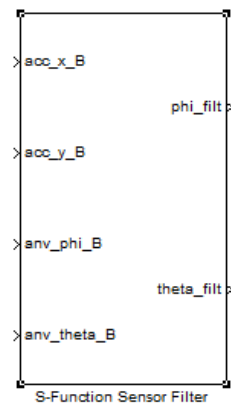


Figure 8.3: Subsystem Sensor Filter

Inputs

Signal	Type	Resolution	Unit
acc_x_B	int16	0.1	m/s ²
acc_y_B	int16	0.1	m/s ²
anv_phi_B	int16	0.01	rad/s
anv_theta_B	int16	0.01	rad/s

Table 8.6: Inputs for Subsystem Sensor Filter

Outputs

Signal	Type	Resolution	Unit
phi_filt	int16	0.0001	rad
theta_filt	int16	0.0001	rad

Table 8.7: Outputs for Subsystem Sensor Filter

Parameters

Signal	Type	Resolution	Unit	Default Value
LPFP	int16	0.01	-	0.05
HPFP	int16	0.01	-	0.95
LPFR	int16	0.01	-	0.05
HPFR	int16	0.01	-	0.95

Table 8.8: Parameters for Subsystem Sensor Filter

Constants

Signal	Value
S_FUNCTION_NAME	sfun_Sensor_Filter_discrete
S_FUNCTION_LEVEL	2
NUMINPORTS	4
NUMOUTPORTS	2
NUMPARAMS	1

Table 8.9: Constants for Subsystem Sensor Filter

References

1. *"Design and Control of an Indoor Micro Quadrotor"*, by Samir Bouabdallah, Pierpaolo Murrieri, Roland Siegwart
2. *"Modelling, Identification and Control of a Quadrotor Helicopter"*, by Tommaso Bresciani, Oct 2008
3. *"Design of a Four-Rotor Aerial Robot"*, by P.Pounds, R.Mahony, P.Hynes & J.Roberts, Proc.2002, Australasian Conference on Robotics & Automation, Auckland, 27-29 November 2002
4. *"Dynamic modeling and nonlinear control strategy for an under actuated quad rotor rotorcraft"*, by Ashfaq Ahmad Mian, Dao-bo Wang, Journal of Zhejiang University SCIENCE A, 2008 9(4):539-545
5. *"Modeling, Simulation and Flight Testing of an Autonomous Quadrotor"*, by Rahul Goel, Sapan M. Shah, Nitin K. Gupta, N. Ananthkrishnan, Proceedings of ICEAE 2009
6. *"Modelling and Control of a Quad-Rotor Robot"*, by Paul Pounds, Robert Mahony, Peter Corke
7. *"Modelling the Dynamics of Flight Control Surfaces Under Actuation Compliances and Losses"*, by Ashok Joshi
8. *"The Balance Filter - A Simple Solution for Integrating Accelerometer and Gyroscope Measurements for a Balancing Platform"*, by Shane Colton, Rev.1: Submitted as a Chief Delphi white paper - June 25, 2007
9. *"An Introduction to the Kalman Filter"*, by Greg Welch and Gary Bishop, UNC-Chapel Hill, TR 95-041, July 24, 2006
10. *"Estimation of attitudes from a low-cost miniaturized inertial platform using Kalman Filter-based sensor fusion algorithm"*, by N Shanta Kumar and T Jann, Sadhana Vol. 29, Part 2, April 2004, pp. 217-235
11. *"Simulink® 7 Writing S-Functions"*, © COPYRIGHT 1998-2007 by The MathWorks, Inc

12. *"C als erste Programmiersprache – Vom Einsteiger zum Profi"*, by Manfred Dausmann, Ulrich Bröckl, Joachim Goll, Rev.5, August 2005, ISBN: 3-8351-0010-6
13. <http://www.digi.com>
14. <http://www.freescale.com>
15. <http://www.jeroenhermans.nl/daedalus>
16. <http://www.mathworks.com>
17. <http://www.mikrokoetter.de>
18. <http://www.netbeans.org>
19. <http://www.pemicro.com>
20. <http://www.wikipedia.de>
21. *Datasheets of electronic parts*