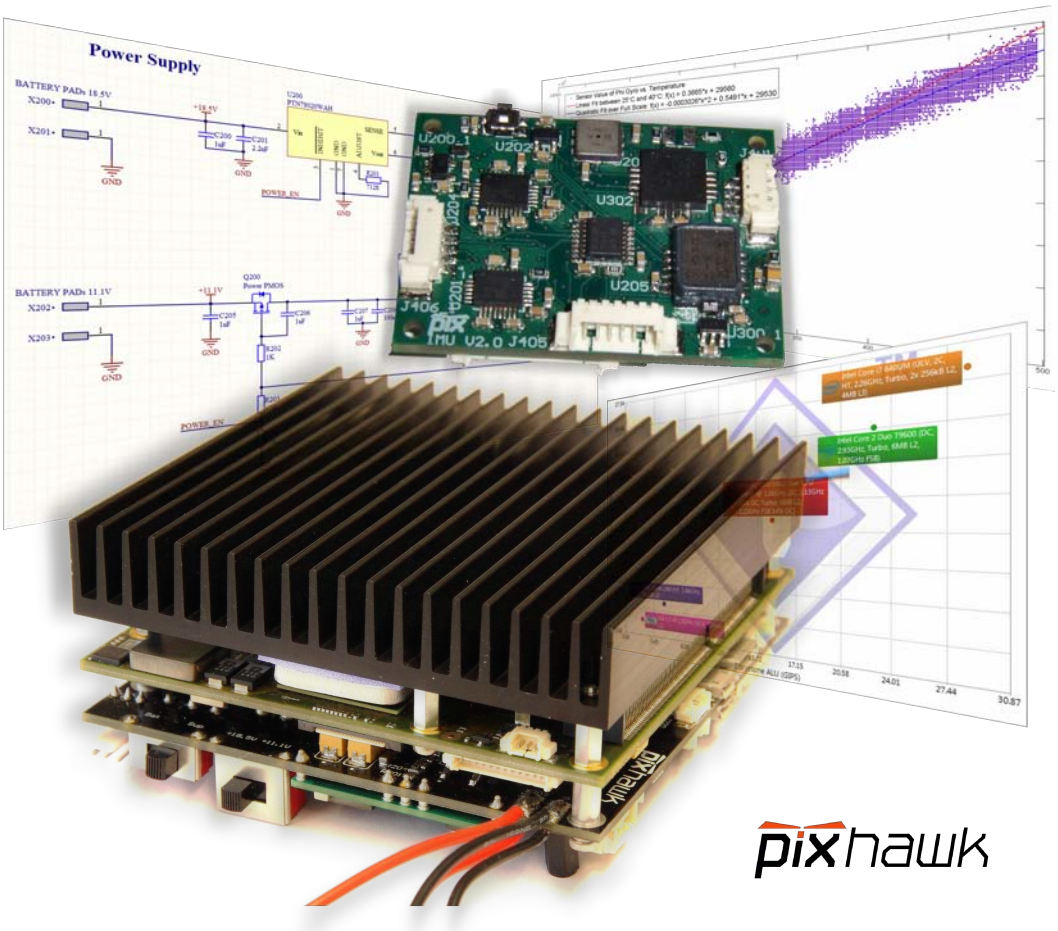
ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Development of Flight Hardware for a Next Generation Autonomous Micro Air Vehicle



Master Thesis IfA / CVG Lab
HS 2009

Christian Dobler

Advisors: Dr. Friedrich Fraundorfer, Lorenz Meier
Professor: Prof. Dr. Manfred Morari

Preface

This master thesis offered me a highly interesting insight into the field of *autonomous Micro Air Vehicles (MAVs)*. I got the opportunity to further improve my electronic engineering knowledge and to gain advantageous practical experience with the most up to data computer technologies.

Specifically I have learnt to use a state-of-the-art toolchain to develop highly integrated and computationally powerful electronics. I have been able to execute the complete electronics development cycle from planning, schematics and layout development, production to hardware testing. In addition I got the opportunity to extend and improve the PIXHAWK code base, which is constantly being evolved by the other team members, for the new flight electronics. In this process I was introduced to a version management toolchain which eases the joint software development.

At this point, I would like to thank the PIXHAWK team leader *Lorenz Meier* for the opportunity to work on such a great project and his support during all the phases of my thesis. I would also like to thank *John Stowers* for his advice and help during the development and manufacturing of the flight electronics. Last but no least I would like to thank *Dr. Friedrich Fraundorfer* and *Prof. Manfred Morari* for being my academic advisors thereby giving me the possibility to conduct my master thesis as a member of the PIXHAWK project team.

Zürich, March 2010

Christian Dobler

Task Description

Introduction

This master thesis is part of the PIXHAWK Micro Air Vehicle project. The goal of the PIXHAWK project is to build a helicopter that will be able to explore indoor and outdoor environments autonomously and identify objects or classes of objects.

Typical usage scenarios are visual inspection of air conditioning systems, and disaster recovery. In disaster recovery, the helicopter will enter semi-collapsed buildings and search for victims needing assistance. Due to the small size and the non-surface-touching drive mechanism, the helicopter can enter locations not accessible by ground robots, dogs or humans.

Due to these use cases, weight reduction and miniaturization as well as operational robustness are critical. The current electronics provide the robustness, but do not comply with the weight and size restrictions. The new electronic system will allow to reduce the helicopter size by 30% from a total length of 160 mm to 120 mm and will also allow to reduce the rotor diameter from currently 340 mm to about 300-260 mm.

Therefore this work is a critical and important step and will help to drive further research on this platform.

Project description

The goal of this master thesis is to create a new flight electronics package for the PIXHAWK platform. The electronics consists mainly of a 12D sensor arrangement, one state of the art MCU and two high-speed single board computers equipped with cameras.

The system is both physically as well as electronically tightly integrated. This allows for hard-realtime operation on the software level and saves weight on the hardware level.

Design task

The new hardware platform includes these design tasks:

- 45% Electronic design
- 45% Embedded software design
- 10% CAD Mechanical design of the electronics package (multiple modules with 3D arrangement)

Platform Requirements

- **Safety Requirements:**
  The electronics package should be maximally safe. As redundancy is not an option due to weight limitations, the most important principle is to reduce the overall failure probability by reducing the number of critical parts. Hence a minimal partcount is desirable.

- **Flight requirements:**
  The electronics should be as light as possible. The PCB dimensions should be minimal to fit within the flight envelope and to reduce mechanical stress. Ultimately a PCB thickness of 1 mm is targeted, although for prototyping 1.6 mm is fine. The necessary components should allow a four layer design.

- **Strategic requirements:**
  As the research platform (the helicopter) is steadily evolving but also constrained very much regarding size, the design should be modular. This allows to fit it perfectly inside the current platform (which requires a 3D arrangement of the PCBs) but to also use it for other use cases, such as:
  - Quadrotor MAVs
  - Fixed Wing MAVs
  - Other inertial applications like sport physiology

- **Sensoric requirements:**
  - 3D Acceleration (milli-G resolution)
  - 3D Angular velocity
  - 3D Earth magnetic field
  - Ultra high-resolution barometric pressure
  - Temperature (approximate, for temperature drift compensation)
  - Battery voltage
  - Optional: Battery current

Work flow

1. **Familiarization with the current PIXHAWK system:** By studying the existing documentation of the current hardware and software, the student has to familiarize himself with the PIXHAWK platform.

2. **Orientation in the fields of acceleration, angular velocity, earth magnetic field and barometric pressure sensors:** Based on a literature research about the state of the art of the priorly mentioned sensors, well-suited components have to be selected.

3. **Time schedule:** The project lasts for 26 weeks. After an orientation time (approx. three weeks) a time schedule including important milestones shall be prepared.

4. **Hardware development:** The Next Generation Visual-Sensory Hardware Platform has to be developed fulfilling the priorly mentioned requirements. Altium Designer is to be used to draw the schematics and footprints and to design the PCBs.

5. **Software development:** The current system software has to be ported to the new hardware platform. This includes rewriting the device drivers for on- and off-chip components. Furthermore the system software has to be extended to make use of the newly integrated components.

6. **Tests and validation:** Functional tests have to be conducted throughout the whole development of the hard- and software. In a final measurement the technical data of the newly developed hardware and the improvement over the current hardware have to be assessed.

Hand-over

- **Report:** Three signed copies of your report have to be handed in by 29. March 2010. This project description should be included at the beginning of your report.

Dr. Friedrich Fraundorfer, Sept. 2009

Abstract

This master thesis deals with the *implementation of a complete flight electronics system for autonomous Micro Air Vehicles (MAVs)* consisting of power supply, orientation sensing platform and on-board computer to execute sophisticated computer vision algorithms.

First of all the logical system architecture had to be designed and the main components had to be evaluated. As a second step the schematics of the new hardware had to be drawn in Altium Designer and the PCBs were layouted. Fabrication files were produced and PCB manufacturers evaluated. The bare PCBs were assembled and logical component groups were electrically tested.

Then a codebase for the embedded software running on the IMU had to be selected and adapted to the new hardware platform. The sensor sampling speed was increased to 200Hz to allow for faster attitude control of the Micro Air Vehicle.

Both functional and quality tests were conducted. The test results were used for the revision of the flight electronics to complete the final hardware design.

Contents

1. Introduction **1**
1.1. The PIXHAWK Project . . . 1
1.2. Previous PIXHAWK Flight Hardware . . . 1
1.3. Challenges . . . 3
1.4. Documentation Overview . . . 3

2. Flight Hardware **5**
2.1. Power Supply . . . 7
2.1.1. Overview . . . 7
2.1.2. Development . . . 7
2.1.3. Manufacturing . . . 8
2.1.4. Specifications . . . 8
2.2. Inertial Measurement Unit . . . 10
2.2.1. Overview . . . 10
2.2.2. Development . . . 13
2.2.2.1. Microcontroller Evaluation . . . 13
2.2.2.2. Accelerometer Evaluation . . . 15
2.2.2.3. Gyroscope Evaluation . . . 16
2.2.2.4. Magnetometer Evaluation . . . 18
2.2.2.5. Pressure Sensor Evaluation . . . 19
2.2.3. Manufacturing . . . 20
2.2.4. Specifications . . . 21
2.3. Baseboard for Gumstix Overo SBC . . . 27
2.3.1. Overview . . . 27
2.3.2. Development . . . 28
2.3.3. Manufacturing . . . 30
2.3.4. Specifications . . . 31
2.4. Baseboard for COM Express SBC . . . 34
2.4.1. Overview . . . 34
2.4.1.1. SBC Form Factor Selection . . . 34
2.4.1.2. Evaluation of COM Express Modules . . . 35
2.4.2. Development . . . 37
2.4.3. Manufacturing . . . 39
2.4.4. Specifications . . . 42
2.4.4.1. Kontron COM Express Module Specifications . . . 42
2.4.4.2. COM Express Baseboard Specifications . . . 43
2.5. Bus-Powered USB-to-UART Adapter . . . 46
2.5.1. Overview . . . 46
2.5.2. Development . . . 46

2.5.3. Manufacturing . . . 46
2.5.4. Specifications . . . 47

3. Embedded Software for IMU **49**
3.1. Codebase Selection . . . 49
3.2. Main Loop - Task Scheduling . . . 50
3.3. Drivers . . . 51
3.3.1. I^2C Subsystem . . . 51
3.3.2. BMP085 Device Driver . . . 54
3.3.3. EEPROM Device Driver . . . 55
3.3.4. I^2C Motor Controller Device Driver . . . 56
3.3.5. SCA3100 Device Driver . . . 57

4. Tests **59**
4.1. Functional Tests . . . 59
4.1.1. Results of Functional Tests of the Power Supply Prototype . . . 59
4.1.2. Results of Functional Tests of the IMU Prototype . . . 59
4.1.3. Results of Functional Tests of the COM Express Baseboard Prototype 60
4.1.4. Results of Functional Tests of the USB-to-UART Adapter . . . 64
4.2. IMU Testing . . . 65
4.2.1. Static Sensor Offset and Noise Distribution Identification . . . 65
4.2.2. Scale Factor Determination for Gyroscopic Angular Velocity Sensors . 67
4.2.3. Temperature Dependency of Gyro Sensors . . . 67
4.3. COM Express SBC Electronics Testing . . . 70
4.3.1. CPU Benchmark . . . 70
4.3.2. Maximum Power Consumption Measurement . . . 73
4.3.3. CPU Temperature Assessment . . . 73

5. Hardware Revision **77**
5.1. Power Supply . . . 77
5.1.1. Hardware Changes . . . 77
5.1.2. Final Specifications . . . 77
5.2. Inertial Measurement Unit . . . 78
5.2.1. Hardware Changes . . . 78
5.2.2. Summary of Changes . . . 78
5.2.3. Final Specifications . . . 79
5.3. Baseboard for Gumstix Overo SBC . . . 80
5.3.1. Hardware Changes . . . 80
5.3.2. Final Specifications . . . 81
5.4. Baseboard for COM Express SBC . . . 81
5.4.1. Hardware Changes . . . 81
5.4.2. Final Specifications . . . 83
5.4.2.1. Kontron COM Express Module Specifications . . . 83
5.4.2.2. COM Express Baseboard Specifications . . . 83
5.5. Bus-Powered USB-to-UART Adapter . . . 84
5.5.1. Final Specifications . . . 84

6. Summary & Outlook **85**
6.1. Summary . . . 85
6.2. Outlook . . . 86

A. Final Schematics **91**
A.1. Power Supply . . . 92
A.2. Inertial Measurement Unit . . . 93
A.3. Baseboard for Gumstix Overo SBC . . . 98
A.4. Baseboard for COM Express SBC . . . 100
A.5. Bus-Powered USB-to-UART Adapter . . . 103

B. Description of Connectors **105**
B.1. Power Supply . . . 105
B.2. Inertial Measurement Unit . . . 107
B.3. Baseboard for Gumstix Overo SBC . . . 110
B.4. Baseboard for COM Express SBC . . . 113
B.4.1. Interface to IMU Board . . . 113
B.4.2. Digital Data Interfaces . . . 114
B.4.3. Various Interfaces . . . 116
B.5. Bus-Powered USB-to-UART Adapter . . . 118

C. C Program Code **119**
C.1. Main Program File (main.c) . . . 119
C.2. Reading the Sensors (sensors.h) . . . 127
C.3. I^2C Subsystem . . . 133
C.3.1. i2c.h . . . 133
C.3.2. i2c.c . . . 135
C.4. BMP085 Pressure Sensor Device Driver . . . 147
C.4.1. bmp085.h . . . 147
C.4.2. bmp085.c . . . 152
C.5. I^2C EEPROM Device Driver . . . 162
C.5.1. eeprom.h . . . 162
C.5.2. eeprom.c . . . 163
C.6. I^2C Motor Controller Device Driver . . . 168
C.6.1. i2c_motor_controller.h . . . 168
C.6.2. i2c_motor_controller.c . . . 169
C.7. SPI Subsystem . . . 172
C.7.1. spi.h . . . 172
C.7.2. spi.c . . . 174
C.8. SCA3100 3D Accelerometer Device Driver . . . 178
C.8.1. sca3100.h . . . 178
C.8.2. sca3100.c . . . 179
C.9. ADS8341 ADC Device Driver . . . 188
C.9.1. ads8341.h . . . 188
C.9.2. ads8341.c . . . 188
C.10. MS2100 3D Magnetometer Device Driver . . . 192
C.10.1. ms2100.h . . . 192

C.10.2. ms2100.c 192
C.11. On-Chip A/D Converter Driver 198
C.11.1. adc.h 198
C.11.2. adc.c 198
C.12. Battery Charge Estimation (battery.h) 201
C.13. Configuration Files 202
C.13.1. conf.h 202
C.13.2. imu_conf_v200.h 205
C.13.3. imu_conf_v210.h 213
C.13.4. pixhawk_2010_mz.h 222
C.13.5. pixhawk_2010_oz.h 222

List of Figures

1.1. The PIXHAWK project logo [1] 1
1.2. Coaxial PIXHAWK helicopter (model 2009) [1] 2
1.3. Mainboard of previous flight electronics [1] 2

2.1. New PIXHAWK Flight Electronics Architecture 6
2.2. Assembled power supply with labels for the main building blocks 7
2.3. Schematics of the power supply prototype 9
2.4. Coordinate frame of PIXHAWK MAVs [1] 10
2.5. Main building blocks of PIXHAWK IMU 11
2.6. Top side of IMU with main building blocks labeled 12
2.7. Bottom side of IMU with main building blocks labeled 12
2.8. Layer stack of the IMU PCB 13
2.9. 3D accelerometer VTI SCA3100-D04 [2] 16
2.10. 2D analog gyroscope Invensense IDG-500 [3] 17
2.11. Digital absolute barometric pressure sensor Bosch BMP085 [4] 20
2.12. Top sheet of the schematics of the IMU prototype 22
2.13. MCU sheet of the schematics of the IMU prototype 23
2.14. Gyroscope and pressure sheet of the schematics of the IMU prototype 24
2.15. Accelerometer and magnetometer sheet of the schematics of the IMU prototype 25
2.16. Connectors sheet of the schematics of the IMU prototype 26
2.17. System architecture of Overo SBC electronics 29
2.18. Top side of Overo baseboard PCB 30
2.19. Schematics sheet of Overo sockets and logic level translators 32
2.20. Schematics sheet of main components on the Overo baseboard 33
2.21. CAD model of the mechanical frame of the new PIXHAWK quadrocopter [1] 36
2.22. Kontron microETXexpress-PC COM Express module [5] 37
2.23. Schematics of 12V power supply design 38
2.24. System architecture of COM Express SBC electronics 40
2.25. PCB of COM Express baseboard 41
2.26. Top side of assembled COM Express baseboard with main blocks labeled . . 41
2.27. Bottom side of assembled COM Express baseboard with main blocks labeled 42
2.28. Schematics sheet of COM Express Type 2 sockets 44
2.29. Schematics sheet of main components on the COM Express baseboard 45
2.30. 4-port USB-to-UART adapter module 46
2.31. Schematics of bus-powered USB-to-UART adapter 48

3.1. Flow chart of infinite while() loop in main.c 52
3.2. Byte fields of i2c_package 53
3.3. Complete I^2C bus transaction to set new RPM value 57
3.4. Complete I^2C bus transaction to get current RPM value 57

4.1. Roll and pitch output of the sensor fusion at 0° inclination 61
4.2. Roll output of the sensor fusion at different inclinations about the X axis . . 62
4.3. Pitch output of the sensor fusion at different inclinations about the Y axis . . 63
4.4. Sensor measurement noise distribution in static reference orientation 66
4.5. Temperature dependency of Phi Gyro 68
4.6. Temperature dependency of Theta Gyro 69
4.7. Temperature dependency of Psi Gyro 69
4.8. Result of arithmetic performance benchmark conducted on the SBC electronics 70
4.9. Result of multimedia performance benchmark conducted on the SBC electronics 71
4.10. Result of arithmetic performance benchmark conducted on Atom based computer 72
4.11. Result of multimedia performance benchmark conducted on Atom based computer 72
4.12. COM Express SBC electronics with Kontron heatsink 73
4.13. Noiseblocker BlackSilent XM2 40x40x10mm fan [6] 74
4.14. System temperatures during CPU burn-in test 75
4.15. Completely assembled COM Express SBC electronics on the scales showing the overall weight with the new cooling solution 76

List of Tables

2.1. Comparison between the TI PTH08T230W and the Emerson LDO06C DC/DC converter modules . . . 8
2.2. Comparison of the NXP LPC2148 and the ST STM32F103 microcontrollers . 14
2.3. Comparison of evaluated digital 3D accelerometers . . . 15
2.4. Specifications of 3D accelerometer VTI SCA3100-D04 . . . 16
2.5. Technical Specifications of the gyroscopes IDG-500 and ISZ-500 . . . 17
2.6. Comparison of evaluated compass sensors . . . 18
2.7. Comparison of digital barometric sensors . . . 20
2.8. Comparison of USB 2.0 hub chips . . . 28
2.9. Comparison of standardized COM form factors . . . 35
2.10. Comparison of two different COM Express modules comprising an Intel Core 2 Duo CPU . . . 36

4.1. Static sensor offsets in reference orientation and sensor accuracy . . . 65
4.2. Gyro sensor values at 45rpm and derived gyro scale factors . . . 67
4.3. Performance increase factor (speed-up) of Core 2 Duo SL9400 vs. Atom 240 . 71
4.4. Power consumption of the PIXHAWK on-board computer under full load . . 73

B.1. Connector Type: 10-pin Molex PicoBlade . . . 105
B.2. Connector Type: 4-pin Molex CLIK Mate . . . 105
B.3. Connector Type: 3-pin Molex PicoBlade . . . 106
B.4. Connector Type: 3-pin Molex PicoBlade . . . 106
B.5. Connector Type: 10-pin Molex PicoBlade . . . 107
B.6. Connector Type: 3-pin Molex PicoBlade . . . 107
B.7. Connector Type: 8-pin Molex PicoBlade . . . 107
B.8. Connector Type: 5-pin Molex PicoBlade . . . 108
B.9. Connector Type: 8-pin Molex PicoBlade . . . 108
B.10. J401: Connector Type: 4-pin Molex PicoBlade . . . 108
B.11. J403: Connector Type: USB Micro-B Receptacle . . . 109
B.12. J405: Connector Type: 6-pin Molex PicoBlade . . . 109
B.13. Connector Type: 4-pin Molex CLIK Mate . . . 110
B.14. Connector Type: 5-pin Molex PicoBlade . . . 110
B.15. Connector Type: 3-pin Molex PicoBlade . . . 110
B.16. Connector Type: 4-pin Molex PicoBlade . . . 111
B.17. Connector Type: 4-pin Molex PicoBlade . . . 111
B.18. Connector Type: 9-pin Molex PicoBlade . . . 111
B.19. J202: Connector Type: 4-pin Molex PicoBlade . . . 112
B.20. J201: Connector Type: USB Micro-AB Receptacle . . . 112
B.21. Connector Type: 10-pin Molex PicoBlade . . . 113
B.22. Connector Type: 3-pin Molex PicoBlade . . . 113

B.23. Connector Type: 4-pin Molex PicoBlade 114
B.24. Connector Type: 4-pin Molex PicoBlade 114
B.25. J220, J221, J227 & J229: Connector Type: 4-pin Molex PicoBlade 114
B.26. J213, J214, J219 & J228: Connector Type: USB Micro-A Receptacle 115
B.27. J224 & J226: Connector Type: USB A Receptacle 115
B.28. Connector Type: 6-pin Molex PicoBlade 115
B.29. Connector Type: 7-pin Molex PicoBlade 115
B.30. Connector Type: 3-pin Molex PicoBlade 116
B.31. Connector Type: 3-pin Molex PicoBlade 116
B.32. Connector Type: 3-pin Molex PicoBlade 116
B.33. Connector Type: 3-pin Molex Fan Header 117
B.34. Connector Type: USB Mini-B Receptacle 118
B.35. Connector Type: 4-pin Molex PicoBlade 118

1. Introduction

This master thesis deals with the *development of flight electronics for a Next Generation Autonomous Micro Air Vehicle (MAV)*. It has been conducted as part of the PIXHAWK project and the developed electronics will be used in upcoming Micro Air Vehicle competitions.

Figure 1.1.: The PIXHAWK project logo [1]

1.1. The PIXHAWK Project

PIXHAWK is a student project team of the Computer Vision and Geometry (CVG) Lab at the ETH Zurich. The goal of the PIXHAWK project is to build a helicopter that will be able to explore indoor and outdoor environments autonomously and identify objects or classes of objects. Typical usage scenarios are visual inspection of air conditioning systems, and disaster recovery. In disaster recovery, the helicopter will enter semi-collapsed buildings and search for victims needing assistance. Due to the small size and the non-surface-touching drive mechanism, the helicopter can enter locations not accessible by ground robots, dogs or humans.

Currently the PIXHAWK helicopters are used at international Micro Air Vehicle competitions. The team has won this years European Micro Air Vehicle 2009 Indoor Autonomy Competition in Delft with the helicopter model shown in figure 1.2.

1.2. Previous PIXHAWK Flight Hardware

The mainboard of the flight electronics mounted on the helicopter in figure 1.2 is shown in graphic 1.3. This mainboard accommodated all the sensors (except the z-gyroscope), two Gumstix Overo single board computers (SBCs) and the power supply. Additionally there is also a separate PCB for the z-axis gyroscope and a backplane containing the connectors for the servo and motor controllers and the socket for the XBee radio module [7].

Figure 1.2.: Coaxial PIXHAWK helicopter (model 2009) [1]

Figure 1.3.: Mainboard of previous flight electronics [1]

All together the electronics are pretty bulky and heavy (61g), mostly due to the incorporation of two Overo sockets on the mainboard and the integration of most of the components on one PCB. This "single-board design" also exhibits the problem, that most of the signals are hard-wired, so that hardly any external peripherals can be attached. This makes this design too inflexible.

On top of this the multi-processor communication interface for the Gumstix Overo SBCs is currently too unstable, so that only one Overo SBC has been used at a time. By removing the second Overo socket a lot of PCB space - and therefore weight - can be saved.

1.3. Challenges

The main challenge of the hardware redesign was to divide the single-board design into several sensible physical units for spatial flexibility when mounting the electronics onto a new helicopter model.

It was also important to provide as many interfaces for different external peripherals as possible. This unfortunately conflicts with the goal of reducing the weight of the electronics, since connectors are usually large compared with other SMD components; i.e. more interfaces means larger PCBs and more weight. Therefore a reasonable compromise between the weight of the electronics and the number of interfaces had to be found.

A further task was to develop a carrier board for a single Gumstix Overo SBC. This carrier board should still give the user the option to connect several modules together for the case that the inter-processor communication becomes more stable and can really be used.

Since the sensor market is evolving rapidly, the sensor selection had to be reviewed. Some sensors have shown good results and should only be replaced, if a newer sensor has got a strong advantage. Other sensors have had several issues and should be replaced if there is another good sensor IC available. This will be detailed in section 2.2.2.

1.4. Documentation Overview

The next chapter (Chapter 2) describes the individual units of the new flight electronics and their development in detail. In Chapter 3 the embedded software running on the *Inertial Measurement Unit (IMU)* is being discussed. Chapter 4 shows the test results and their analysis resulting in the final hardware revision (chapter 5). Finally there is a short summary and an outlook of how the flight electronics and the embedded software could be improved further.

2. Flight Hardware

This chapter describes the development, the manufacturing and the specifications of the new PIXHAWK flight hardware in detail. The selection of the main components to implement the requested functionality is also discussed here. The PCBs of the flight electronics have been designed with Altium Designer Summer 2009, a state-of-the-art PCB development tool suite.

The flight hardware described on the following section resembles the first prototype of the new PIXHAWK flight electronics and not the final version. Changes made to the hardware shown in this chapter will be discussed in chapter 5 (hardware revision).

The single-board design of the previous flight electronics has been divided into three separate physical units:

- Power Supply
- Inertial Measurement Unit (IMU)
- Single Board Computer (SBC) Baseboard

This design is shown in figure 2.1 and allows for more spatial flexibility when mounting the electronics on a specific helicopter model. It also gives the possibility to integrate more interfaces to external components. This means that peripherals like cameras, GPS modules, infrared sensors, etc. can be attached to the flight electronics when needed to improve the autonomy of the Micro Air Vehicle.

These units will be described in detail in the following sections.

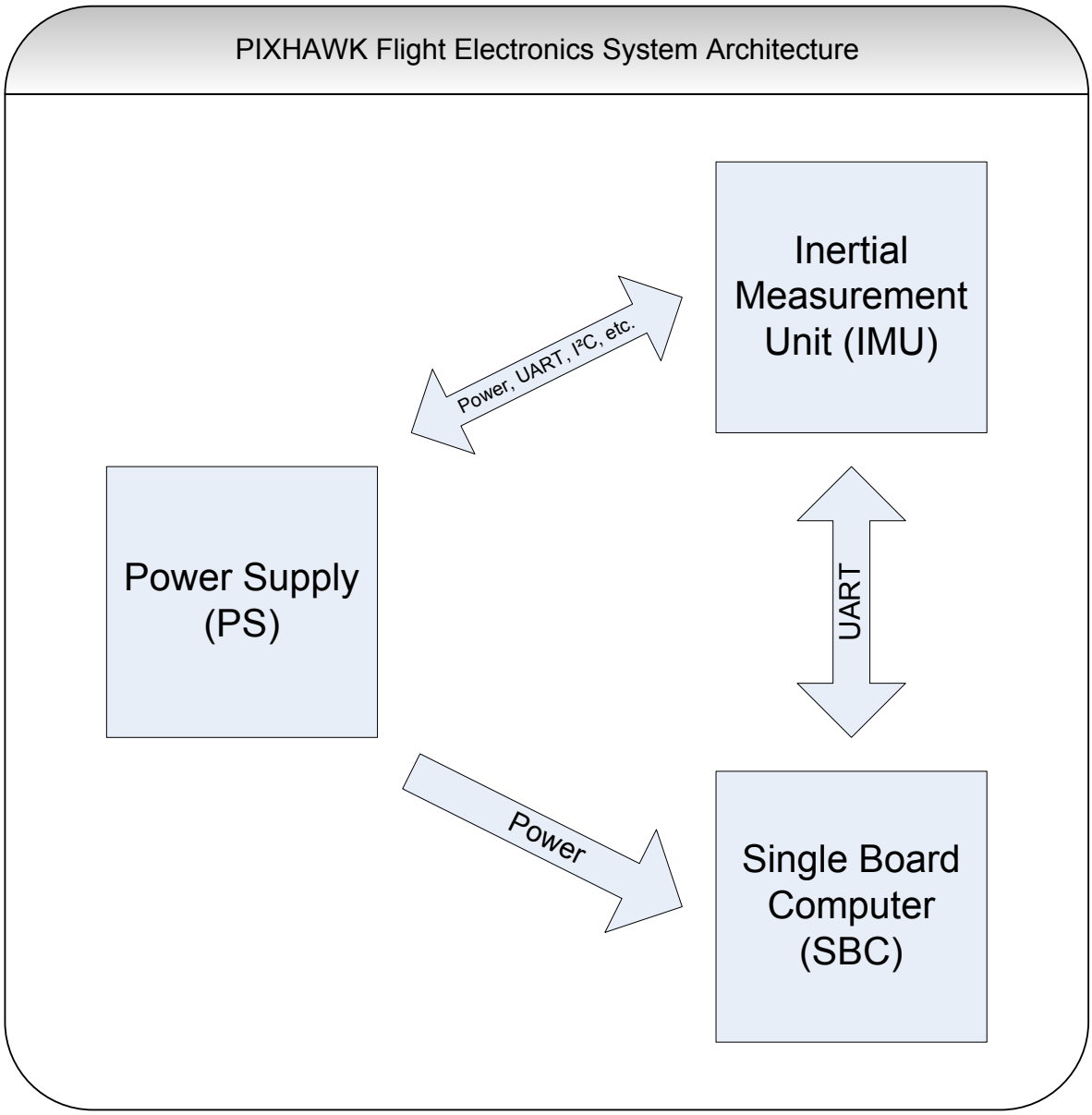


Figure 2.1.: New PIXHAWK Flight Electronics Architecture

2.1. Power Supply

2.1.1. Overview

Graphic 2.3 shows the assembled prototype of the power supply with markings for the main building blocks. The schematics of the prototype are shown in the figure 2.3.

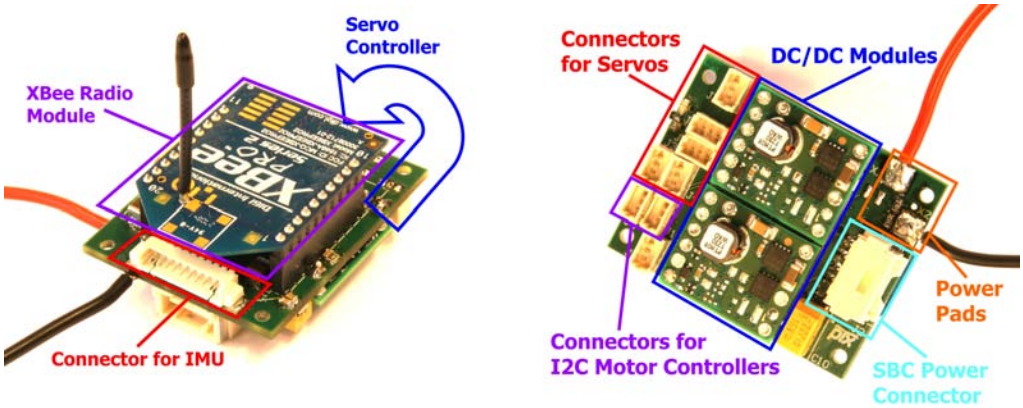


Figure 2.2.: Assembled power supply with labels for the main building blocks

2.1.2. Development

The *concept* behind the design of the power supply was to integrate all components consuming a lot of electrical power (except for the single board computer) on this board. This reduces the amount of electrical noise injected into the sensor platform. Therefore the servo controller, the servo connectors and the connectors for the new, digital I^2C motor controllers are also located on this board beside the power supply. For spatial reasons also the socket for a XBee Pro radio module is placed on this unit. The XBee Pro [7] is a 2.4 GHz ZigBee based radio module which is used for bidirectional communication with the PIXHAWK groundstation.

The power supply converts the battery voltage of nominally 11.1V (3-cell lithium polymer battery pack) by using two Texas Instruments PTH08T230W DC/DC converter modules [8] to 5.0V and 3.7V. Other DC/DC converter modules have been evaluated. The final selection has been made between the TI PTH08T230W and the LDO06C [9] from Emerson. Table 2.1 compares the most important features of these DC/DC converter modules. The Texas Instruments PTH08T230W DC/DC converter module has been selected mostly due to availability reasons and because it complies to the POLA [10] standard which increases the interchangeability with devices from other manufacturers.

The 3.7V output is necessary to power the Gumstix Overo SBC and can also be used to create low-swing 3.3V power outputs by using low-dropout regulators without a large power loss. Each power rail can provide up to 6A of current.

The servo controller is implemented as a 5 Stage Johnson Decade Counter NXP HEF4017B that converts the PPM control signal coming from the microcontroller to one PWM signal

TI PTH08T230W [8]	**Emerson LDO06C [9]**
6A max. output current	6A max. output current
Weight: 2.5g	Weight: 1.9g
Dimensions: 18.92mm x 15.75mm	Dimensions: 16.51mm x 10.41mm
complies to POLA standard	not standardized
good availability	only available on order

Table 2.1.: Comparison between the TI PTH08T230W and the Emerson LDO06C DC/DC converter modules

per servo. This solution has been adopted from the previous PIXHAWK flight electronics. A detailed documentation of the signal conversion process can be found in [11].

2.1.3. Manufacturing

Since the PCB manufacturer, that had been used for the previous PCB orders, was unable to deliver enough manufacturing precision for the new design, other manufacturers had to be evaluated. Two prototypes of the power supply PCB have been manufactured by Multi PCB in Germany.

The PCBs have been hand-soldered and functional tests have been conducted block by block while assembling the unit.

2.1.4. Specifications

- **Power Input:** 6.0V - 14V DC
- **Power Output:**
  - max. 6A @ 5.0V
  - max. 6A @ 3.7V
- **XBee Pro:**
  - 2.4 GHz wireless module
  - based on ZigBee
  - max. data rate: 115200 bits/s (57600 bits/s bidirectional)
  - max. current consumption: 295 mA @ 3.3V (transmit current)
  - range: 100 m indoors, 1.6 km outdoors
- **Servo Controller:**
  - 5 Stage Johnson Decade Counter NXP HEF4017B for PWM control of servos
  - connectors for up to 6 servos
- **Motor Controllers:** connectors for 2 digital I^2C motor controllers
- **Size of PCB:** 40 x 32 mm
- **Weight (completely assembled):** 18g (with XBee module: 22g)

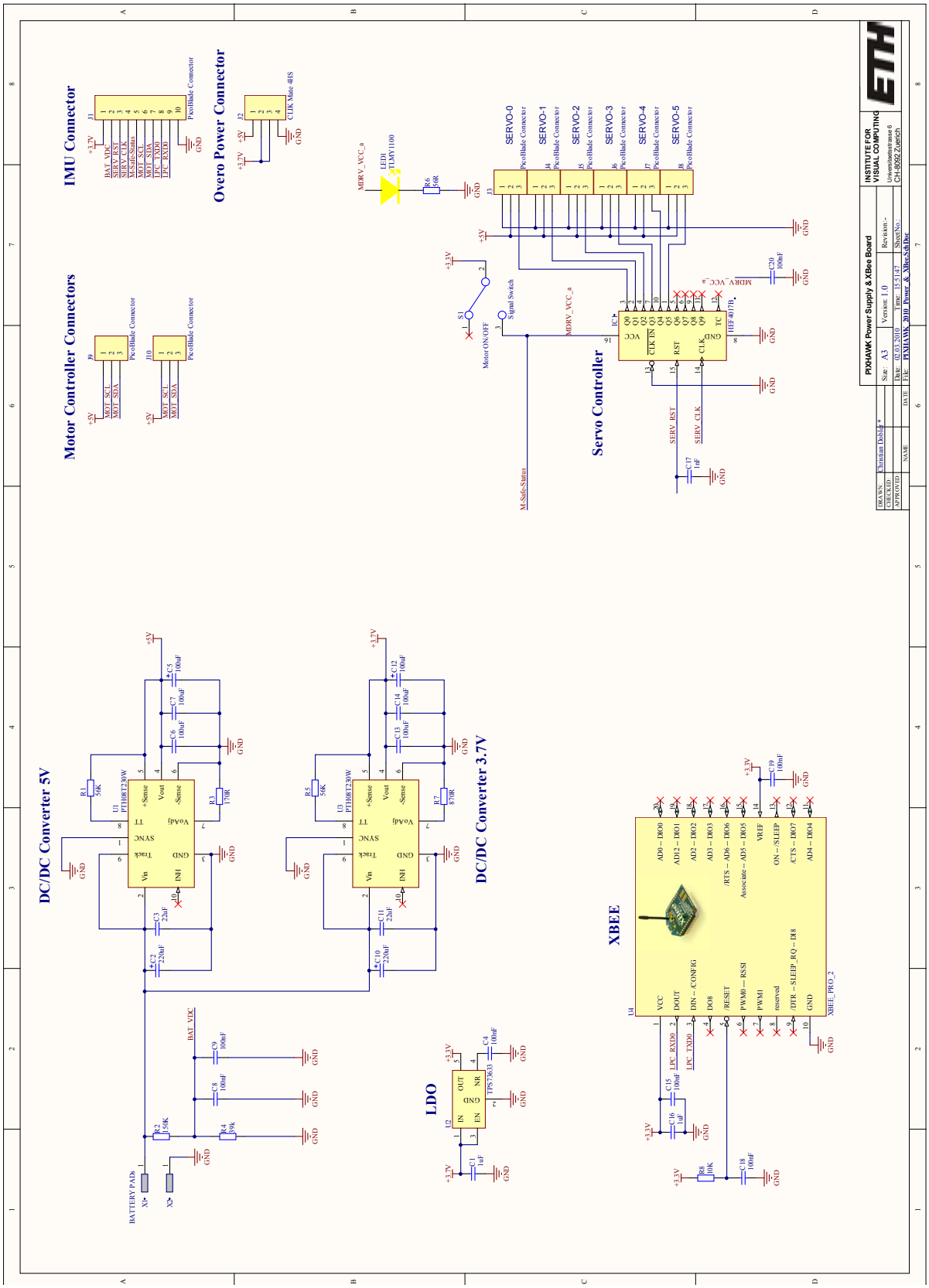


Figure 2.3.: Schematics of the power supply prototype

2.2. Inertial Measurement Unit

2.2.1. Overview

The *Inertial Measurement Unit (IMU)*[1], shown in figures 2.6 and 2.7, contains all sensors necessary to track the orientation of the MAV. This board comprises a *3D accelerometer*, a *2D gyroscope* (X and Y axis) and a *1D gyroscope* (Z axis). The measurements from these sensors are used by the sensor fusion process to determine the angular position of the MAV about the horizontal axes (X and Y axis) [12, 13]. This is called the *roll (angular position about X axis = Φ (Phi) angle)* and the *pitch (angular position about Y axis = Θ (Theta) angle)*. The coordinate frame used for the PIXHAWK MAVs is shown in figure 2.4.

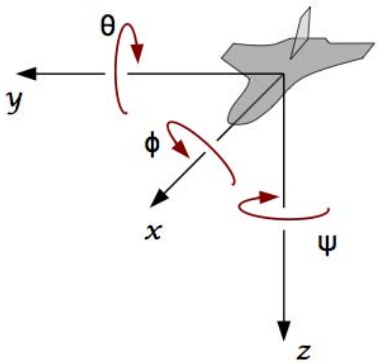


Figure 2.4.: Coordinate frame of PIXHAWK MAVs [1]

Also a *3D magnetosensor* is integrated on this board usable as a digital compass to sense the earth's magnetic field. The sensor fusion uses these measurements combined with the data from the gyroscopes to determine the angular position about the Z axis [12, 13]. This is called the *yaw or Ψ (Psi) angle*. All of these sensors are manufactured nowadays in MEMS (micro electromechanical systems) technology, which makes it possible to integrate complex electromechanical systems in tiny IC packages.

For outdoor applications the IMU also comprises a *digital barometric pressure sensor* to determine the flying altitude of the MAV more accurately than a GPS module would be able to.

The sensor fusion and control can and is done on the IMU by the embedded software running on an ARM 7 based microcontroller contained on the board. The sensor calibration data can be stored in a EEPROM. The architecture of the new IMU is depicted in figure 2.5 and figures 2.6 and 2.7 show the assembled IMU with the main building blocks marked. The schematics of the IMU can be found in figures 2.12 to 2.16 at the end of this section.

[1] The term "IMU" might be somewhat misleading, since the exact definition of an IMU only comprises a 3D accelerometer and a 3D gyroscope. This board features additionally a 3D magnetometer, which makes it an *Attitude and Heading Reference System (AHRS)*, and a barometric pressure sensor to track the flying altitude of the MAV. There is no fixed term for this sensor arrangement and IMU is commonly used in the field of MAV research for an orientation measurement systems. That is why this board is called "IMU".

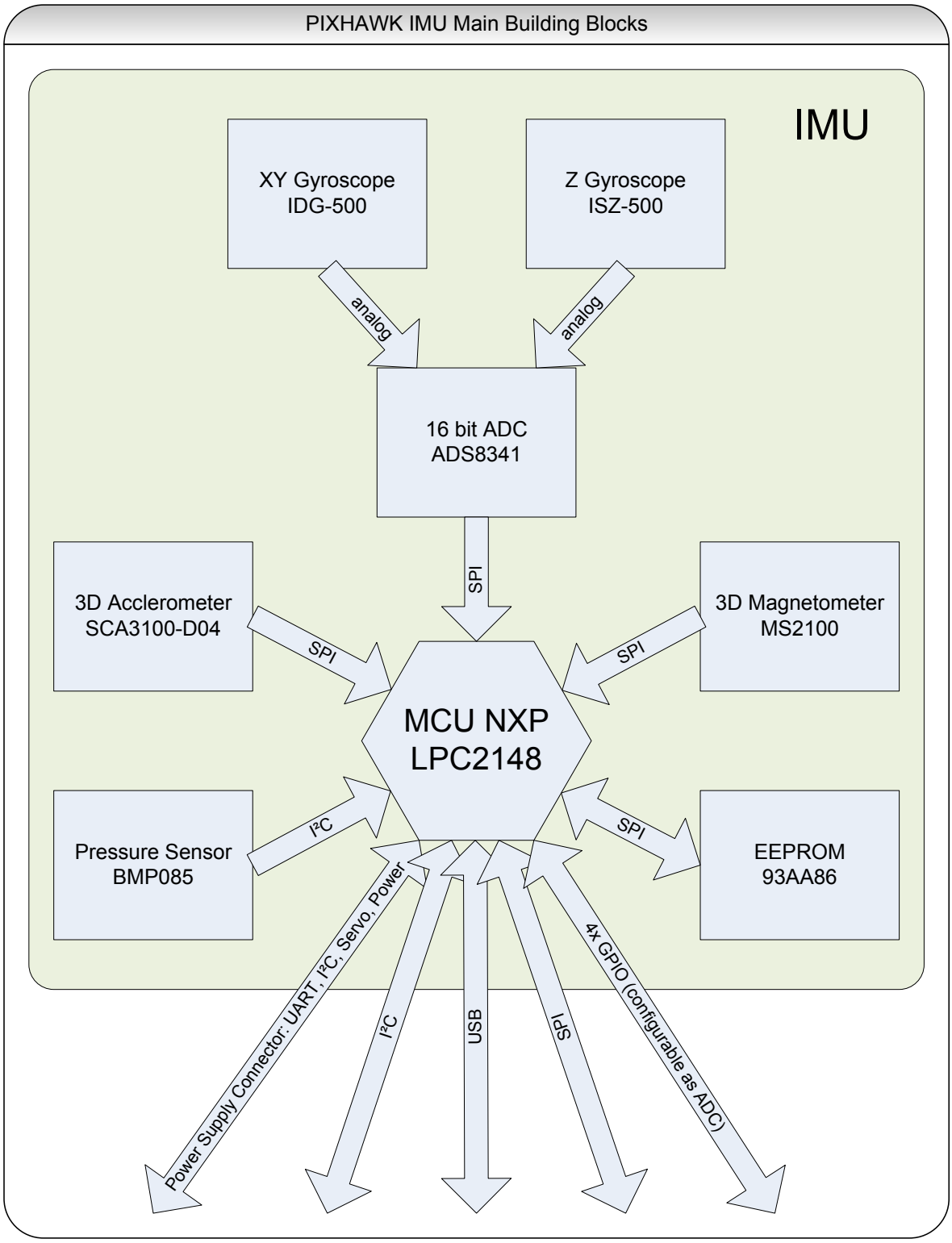


Figure 2.5.: Main building blocks of PIXHAWK IMU

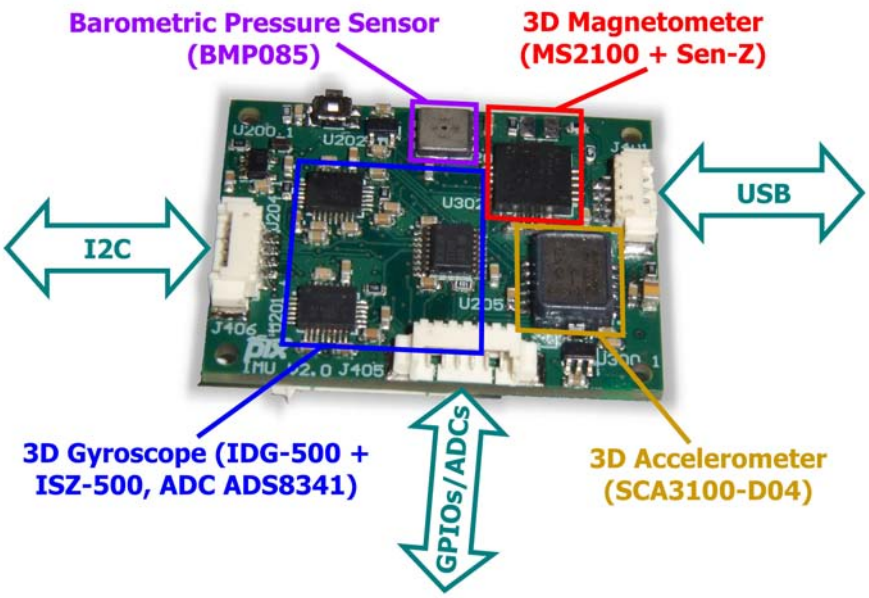


Figure 2.6.: Top side of IMU with main building blocks labeled

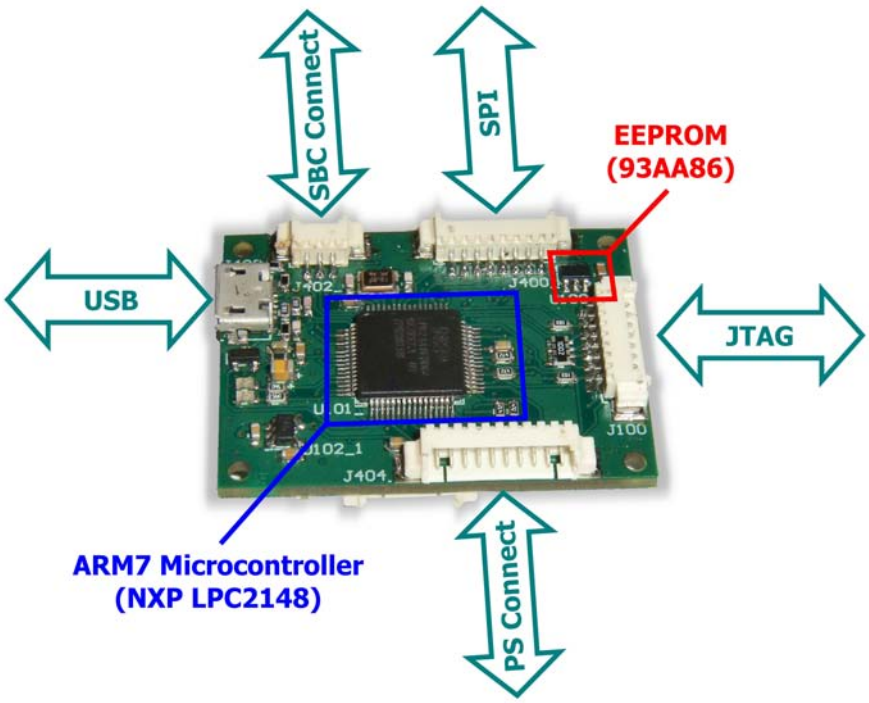


Figure 2.7.: Bottom side of IMU with main building blocks labeled

2.2.2. Development

This section deals with the *development of the IMU*, i.e. the evaluation and selection process for the different components will be described here.

An important task in the new design of the IMU beside the part evaluation was to reduce the overall part count. This should be done by a transition from analog sensors to digital sensors wherever possible which again has influence on the component selection. Also the large analog voltage supply filtering circuits should be reduced. To achieve this a new approach to supplying the sensors with a clean and noiseless power signal had to be introduced. The new approach was to place a dedicated LDO per component on the design, which effectively block noise on the input and produce a low voltage ripple on the output. These LDOs are placed very close to the associated sensor in the layout to reduce the amount of noise injected into the power signal.

To reduce the electrical noise injected into the sensors further, all sensors are placed on one layer which is separated by first a ground layer and then a power layer from the rest of the components on the second outer layer. Figure 2.8 shows the layer stack of the IMU PCB. Also all parts consuming a lot of electrical power, e.g. servo controller, are moved to the power supply board and only the control signals are generated on the IMU by the MCU. This reduces the noise level on the sensor platform again.

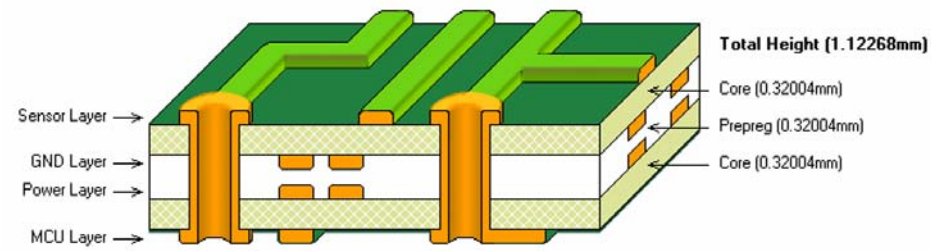


Figure 2.8.: Layer stack of the IMU PCB

Another task was to establish pin compatibility with the Twog board from the Paparazzi (PPZ) project [14]. This platform has resembled a quasi-standard in the field of MAV research. The establishment of hardware compatibility with this platform should facilitate the transition to the PIXHAWK platform for other researchers. This is an attempt to offer a feasible alternative to the Paparazzi platform to the MAV research community.

The following subsections will give detailed information about the part selection for the most important components.

2.2.2.1. Microcontroller Evaluation

The microcontroller (MCU) used in the previous PIXHAWK flight electronics has been a NXP LPC2148 [15], which is based on the ARM 7 architecture. The only real drawback of this microcontroller is, that the resolution of the on-chip ADCs is limited to 10 bits, which is not enough for our application. This makes the use of an external A/D converter for the

analog components inevitable. Newer microcontroller architectures provide 12 bit ADCs on-chip which would remove the need for an external ADC and therefore save PCB space and weight. The higher processing speed of new architectures is not needed because the current software loads the LPC2148 core only to 15%.

Since all the embedded code (including drivers) has been developed for the ARM 7 architecture, only very similar architectures can be used to not generate a huge amount of software porting work. For this reason only MCUs based on other ARM architectures have been evaluated, namely ARM 9 and Cortex M3 based devices. None of the ARM 9 based devices was suitable for our application because they are something in between microcontrollers and microprocessors, i.e. their IC packages are too large. The STM32F103 [16] from ST Microelectronics has been selected from the available Cortex M3 based MCUs as the most promising candidate to be evaluated versus the LPC2148. Table 2.2 shows a comparison of these two MCUs.

	NXP LPC2148 [15]	**ST STM32F103 [16]**
Architecture	ARM 7	ARM Cortex M3
Advantages	64 pin package	64 pin package
	current PIXHAWK software runs on this (also Booz/Wasp/PPZ)	3 UARTs
	decent USB bootloader for Windows and Linux available	12 bit ADC
	Makefile and Eclipse project available, click to compile works	CAN interface
	current PIXHAWK CPU load is only 15%, no need for faster processor	
Disadvantages	only 2 UARTs	toolchain support unknown
	only 10 bit ADC	bootloader status unknown
	no CAN interface	software development, incl. drivers, from scratch needed

Table 2.2.: Comparison of the NXP LPC2148 and the ST STM32F103 microcontrollers

As table 2.2 shows, the STM32F103 offers slightly better on-chip peripherals. But more importantly research on the Cortex M3 architecture [17] has shown, that it would need a large effort to port the existing embedded software to this architecture. The Cortex M3 is a completely new architecture with a new instruction set and new on-chip peripherals. This means that all the drivers would need to be reprogrammed. Also no information on available bootloaders and an open-source development toolchain could be found.

Because the advantages in performance of the Cortex M3 architecture are not vital to us, it was decided to use the NXP LPC2148 as before in the new IMU design.

2.2.2.2. Accelerometer Evaluation

The accelerometer is needed to determine the pitch and the roll angles of the MAV in static flight attitudes. As the dynamics of the flight attitude increase, the occurring accelerations and high frequency measurement noise distort the angle determination by the signal fusion [18]. The gyroscopes will aid the signal fusion in calculating the orientation in dynamic situations.

For identifying the orientation of the MAV the used accelerometer must only be able to measure accelerations between ±1g since the norm of the acceleration vector must always result in 1g in static situations. If the absolute value of the acceleration vector is of interest, a larger measurement range might be necessary depending on the application. Relying on experience the acceleration values of an MAV will not exceed ±2g in any controllable flight situation.

Therefore a variety of digital 3D accelerometers with a full scale measurement range of at least ±2g has been evaluated. The most important characteristics of these sensors are listed in table 2.3.

	Measurement Range	Max. Resolution	Size
SCA3000-D01 [19]	±2g	0.0074 m/s^2	7x7x1.8 mm
ADXL345 [20]	±16g	0.019 m/s^2	3x5x1 mm
MMA7455L [21]	±8g	0.019 m/s^2	3x5x1 mm
LIS302DL [22]	±8g	0.077 m/s^2	3x5x0.9 mm
LIS3LV02DQ [23]	±6g	0.0098 m/s^2	7x7x1.8 mm
IME-3000	±24g	0.0006 m/s^2	3x3x? mm

Table 2.3.: Comparison of evaluated digital 3D accelerometers

The IME-3000 looks like the clear winner in this comparison and it was in fact very promising. It's tiny, sports the highest resolution of all sensors in this comparison and it even contains three 16 bit ADCs to attach other sensors. Sadly it was not available yet at the time of the evaluation. An inquiry to the manufacturer Invensense was replied to with the information, that the IME-3000 would be available during November 2009. But the manufacturer was not able to provide a complete datasheet and therefore the IME-3000 had to be removed from the evaluation list. In November this decision was proven right because Invensense had removed the IME-3000 product page without any notification or replacement.

Since the resolution is the most important factor for us, the SCA3000-D01 from VTI Technologies looks like the next best option. This sensor has already been used on the previous PIXHAWK electronics and never had any major issues. Another advantage of this chip is that the driver already existed in the PIXHAWK codebase. For these reasons the SCA3000-D01 was selected as the 3D accelerometer to use on the new IMU.

Just at the end of the schematics design when collecting all the part numbers for the bill of material, VTI had suddenly declared the complete SCA3000 sensor series as obsolete. Since an obsolete part should not be integrated on a new design, a replacement that would not

necessitate a lot of other changes had to be found quickly. Beside declaring the SCA3000-D01 as obsolete, VTI had also introduced a replacement product, named SCA3100-D04 [2] (see figure 2.9). Even though this replacement is slightly bigger and its resolution has decreased a bit, it still does well compared to the other sensors. Therefore the schematics have been adapted to newly incorporate the SCA3100-D04. The main characteristics of this sensor can be found in table 2.4.

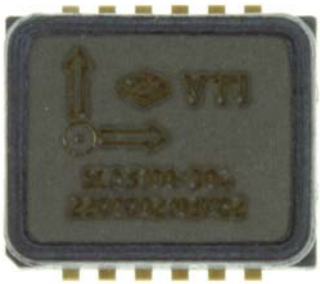


Figure 2.9.: 3D accelerometer VTI SCA3100-D04 [2]

	VTI SCA3100-D04
Supply Voltage	3.3V
Measurement Range	±2g
Max. Resolution	0.0109 m/s^2
Size	7.6x3.3x8.6 mm
Digital Interface	SPI
Max. Output data rate	2000Hz

Table 2.4.: Specifications of 3D accelerometer VTI SCA3100-D04

2.2.2.3. Gyroscope Evaluation

A 3D gyroscope measures the angular velocity along the three axes of its local coordinate frame. This means that in static situations the output from a gyroscope is always zero (or whatever its static offset is) independent of its orientation. The sensor fusion can therefore only use the gyro measurements for short-term angular changes by continuously integrating the measurements [12]. In this process small measurement errors can produce large errors in the orientation output due to integration. These integration errors can be removed during temporary static situations by using the data from the accelerometer.

At the time of the sensor evaluation process no gyroscopes comprising three axis in one chip were available. Because the XY-gyroscope IDG-500 [3] (see figure 2.10) from Invensense is tiny and has been working well on the previous flight electronics, there was no motivation to reevaluate this sensor. The solution of the previous electronics to measure the velocity about the Z-axis was an extra PCB containing another IDG-500 that was mounted vertically

onto the mainboard. This solution had been chosen since no Z-gyroscope consistent with the IDG-500 was available at the time of development. But it was not satisfying because the vertical PCB could not be mounted accurately and suffered from a lot of vibrations during flight.

Figure 2.10.: 2D analog gyroscope Invensense IDG-500 [3]

Meanwhile Invensense had released the Z-axis gyroscope ISZ-500 [24] which integrates perfectly with the IDG-500 and has the same specifications. For this reason the ISZ-500 was selected as the Z-gyro of the new IMU. Table 2.5 lists the main specifications of the IDG-500 and ISZ-500.

	IDG-500 [3]	**ISZ-500 [24]**
Manufacturer	Invensense	Invensense
Axes	X, Y	Z
Output	analog	analog
Measurement Range	±500 °/s	±500 °/s
Sensitivity	2.0 mV/°/s	2.0 mV/°/s
Size	4x5x1.2 mm	4x5x1.2 mm

Table 2.5.: Technical Specifications of the gyroscopes IDG-500 and ISZ-500

As it can be seen from table 2.5, the output of these sensors is analog. Since the MCU only integrates 10-bit ADCs, an external ADC is necessary to convert the gyro measurements more accurately. Th part number of the external ADC used on the IMU is ADS8341E [25]. This is a four channel 16 bit ADC from Texas Instruments providing an SPI interface as the digital interface to the MCU. Three ADC channels are used for the angular velocity measurement while the fourth channel is connected to the temperature sensor built into the IDG-500 to assess the temperature of the gyroscopes. The gyroscopic measurements are strongly susceptible to temperature changes that have to be compensated in software.

2.2.2.4. Magnetometer Evaluation

A 3D magnetometer is implemented on the IMU for tracking the yaw (Ψ, Psi) angle of the MAV. The X- and Y-axis measurements are used by the sensor fusion to determine the current value of the yaw angle while the Z component is used for tilt compensation [18]. This allows for a correct heading computation even if the IMU is not held flat.

There is a wide variety of magnetometers for measuring magnetic fields of different strengths. As we want to measure the earth's magnetic field, the main criterion for the selection is that the magnetometer is able to sense the earth's magnetic field whose intensity is about 48μT in middle Europe. Such magnetometers are also called compass sensors. Table 2.6 lists the main characteristics of the evaluated compass sensor ICs.

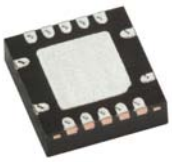
	MS2100 [26]	**HMC5843 [27]**	**AK8973 [28]**	**HMC6052 [29]**
				
Manufacturer	PNI	Honeywell	AKM	Honeywell
No. of Axes	3 (ext. Z-sensor)	3	3	2
Output	digital (SPI)	digital (I^2C)	digital (I^2C)	analog
Measurement Range	$\pm$1100μT	$\pm$450μT	NA	$\pm$200μT
Max. Resolution	0.055μT	0.034μT	NA (8 bit)	-
Bandwidth	>10kHz	10kHz	NA	1kHz
Size	7x7x1.35 mm	4x4x1.3 mm	4x4x0.7 mm	3.5x3.5x0.8 mm
IC Package	20 pin LCC	20 pin LCC	16 pin QFN	14 pin QFN

Table 2.6.: Comparison of evaluated compass sensors

From table 2.6 it can be seen that the AK8973 and the HMC6052 cannot compete with the other two sensors regarding their specifications. The reason for evaluating these sensors is that their IC packages are much better suited for hand-soldering than those of the MS2100 and the HMC5843 (this issue will be discussed in the paragraphs below).

The AK8973 is the compass IC used in the Apple iPhone and other smartphones. Its resolution is limited to 8 bits which is really low compared to the 12 bit resolution of the other two digital sensors. Also it is not available in small quantities and the datasheet is lacking important information.

The HMC6052 only comprises a dual-axis sensing element which makes tilt compensation impossible. Furthermore it is an analog sensor and would therefore additionally require analog to digital conversion.

Based on the technical specifications the MS2100 and the HMC5843 perform very similar (except that the MS2100 is a bit bigger and additionally needs an external Z sensor coil). But these sensors exhibit a strong difference in how the measurements can be performed. The HMC5843 constantly performs internal oversampling to increase the resolution of its measurements and new measurement data can only be read out at a maximum rate of 50Hz. The MS2100 also does internal oversampling but the data output rate can be increased up to nearly 5kHz by setting the sensor's configuration appropriately. This automatically reduces the resolution of each measurement but it can be increased again by manually doing the oversampling on the MCU. So the main difference between the HMC5843 and the MS2100 is that the MS2100 offers the programmer the choice to do the oversampling manually.

Why would we want to do the oversampling manually? The reason is that this IMU will be mounted on a MAV containing electromotors. The strong electromagnetic field generated by the motors is superimposed on the earth's magnetic field and the magnetic field measurement is dominated by the field of the motors. The magnetic field of the motors has a maximum frequency of 300Hz. Based on the sampling theorem the rotary field of the motors can be completely reconstructed from the digital measurements if the magnetosensor is sampled at a rate $\geq$600Hz. By appropriately low-pass filtering the signal the rotating field components can be removed from the measurements yielding the direction of the earth's magnetic field. If we can do this manually on the MCU, we can be sure that the low-pass filter is designed correctly. If the oversampling is done inside the magnetosensor, we can only hope that high frequencies are filtered correctly.

For this reason the MS2100 had already been used on the previous PIXHAWK flight electronics. The biggest issue with this chip is that it can hardly be soldered by hand (only 1 out 10 chips could be mounted correctly) which really is a problem for the prototyping process. But we have been able to get an arrangement with the Electronics Laboratory at ETH Zurich that we can use their reflow oven to assemble our prototypes. It was assumed that this change in the manufacturing process would eliminate the mounting issue. Therefore the MS2100 was also selected for the new PIXHAWK IMU.

2.2.2.5. Pressure Sensor Evaluation

To determine the flying altitude of the MAV in outdoor applications a barometric absolute pressure sensor is integrated on the PIXHAWK IMU. On the previous flight electronics the analog pressure sensor MPXH6115 from Freescale had been used. This sensor is huge compared to the other sensors on the IMU and additionally needs a complex analog amplification circuit which consumes even more PCB space. The goal of this sensor evaluation was to find a small digital barometric pressure sensor as a replacement for the MPXH6115.

Only two digital SMD pressure sensors could be found. The main characteristics of these barometric sensors are listed in table 2.7.

Table 2.7 shows that both sensors are small and very accurate with a small advantage for the BMP085 (see figure 2.11). Just in the aspect of absolute pressure accuracy the SCP1000-D01 performs slightly better. But this aspect is not important to us because only the difference

	BMP085 [4]	**SCP1000-D01 [30]**
Manufacturer	Bosch Sensortec	VTI Technologies
Measurement Range	300 - 1100 hPa	300 - 1200 hPa
Max. Resolution	0.01 hPa $\approx$ 0.08 m	0.015 hPa $\approx$ 0.12 m
Absolute Accuracy	$\pm$2.5 hPa	$\pm$1.5 hPa
Relative Accuracy	$\pm$0.2 hPa	$\pm$0.5 hPa
Max. Conversion Rate	222 Hz	9 Hz
Chip Size	5x5x1.2mm	6.1x6.1x1.7mm

Table 2.7.: Comparison of digital barometric sensors

between the flying altitude and the ground is of interest. Therefore the BMP085 from Bosch Sensortec was selected for the new IMU. As VTI Technology has declared the SCP1000-D01 obsolete at the end of October 2009 without presenting a replacement part, this selection would now be inevitable.

Figure 2.11.: Digital absolute barometric pressure sensor Bosch BMP085 [4]

A nice feature of the BMP085 is that it has a temperature sensor integrated in the chip. This is necessary to scale the pressure measurement correctly and can also be used for temperature compensation of other sensors on the IMU.

2.2.3. Manufacturing

Two prototypes of the PIXHAWK IMU PCB have been manufactured by Multi PCB in Germany. The top side (sensor layer) components have been assembled in the reflow oven of the Electronics Laboratory at ETH Zurich while the bottom side has been hand-soldered.

2.2.4. Specifications

- **Power Input:** 3.7V DC
- **Power Consumption:** max. 98mA @ 3.7V ≈ 363mW
- **Microcontroller:**
  - Part Number: NXP LPC2148 [15]
  - 16-bit/32-bit ARM7TDMI-S microcontroller
  - Max. Clock Frequency: 60 MHz
- **Sensing Capabilities:**
  - 3D Linear Acceleration (VTI SCA3100-D04 [2]):
    - Resolution: 1.11 mg
    - Range: ±2g
  - 3D Angular Velocity (Invensense IDG-500 [3] and ISZ-500 [24]):
    - Resolution: 0.1 deg/s
    - Range: ±500 deg/s
  - 3D Earth magnetic field (PNI MS2100 [26]):
    - Resolution: 0.055 μT
    - Range: ±1100 μT
  - 1D Barometric Pressure (Bosch BMP085 [4]):
    - Resolution: 0.08 m
    - Range: -500 m at 110 kPa (101.325 kPa at mean sea level) to about +9000 m at 30 kPa
  - 1D Temperature (Bosch BMP085 [4]):
    - Resolution: 0.1 °C
    - Range: -20 - 65 °C
- **External Interfaces:**
  - 1x USB Micro-B
  - 1x SPI with 3 chip select lines
  - 1x I^2C plus one GPIO pin
  - 4x GPIO pin configurable as ADC input
  - 1x UART interface to SBC
  - 1x interface to power supply (contains signals for servo / motor controllers, UART interface to XBee, battery sensing and power lines)
- **Size of PCB:** 40 x 30.2 mm
- **Weight (completely assembled):** 8g

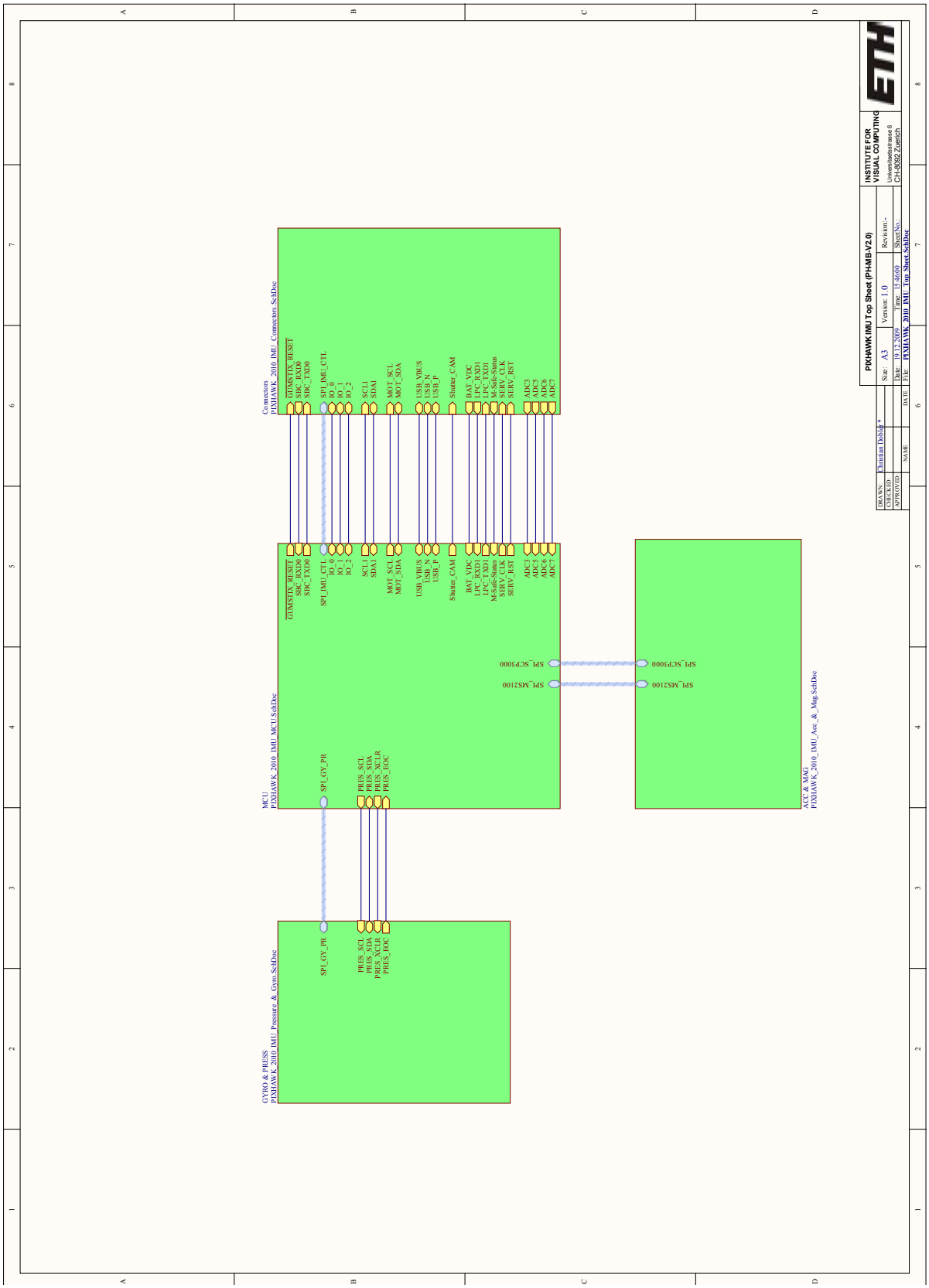


Figure 2.12.: Top sheet of the schematics of the IMU prototype

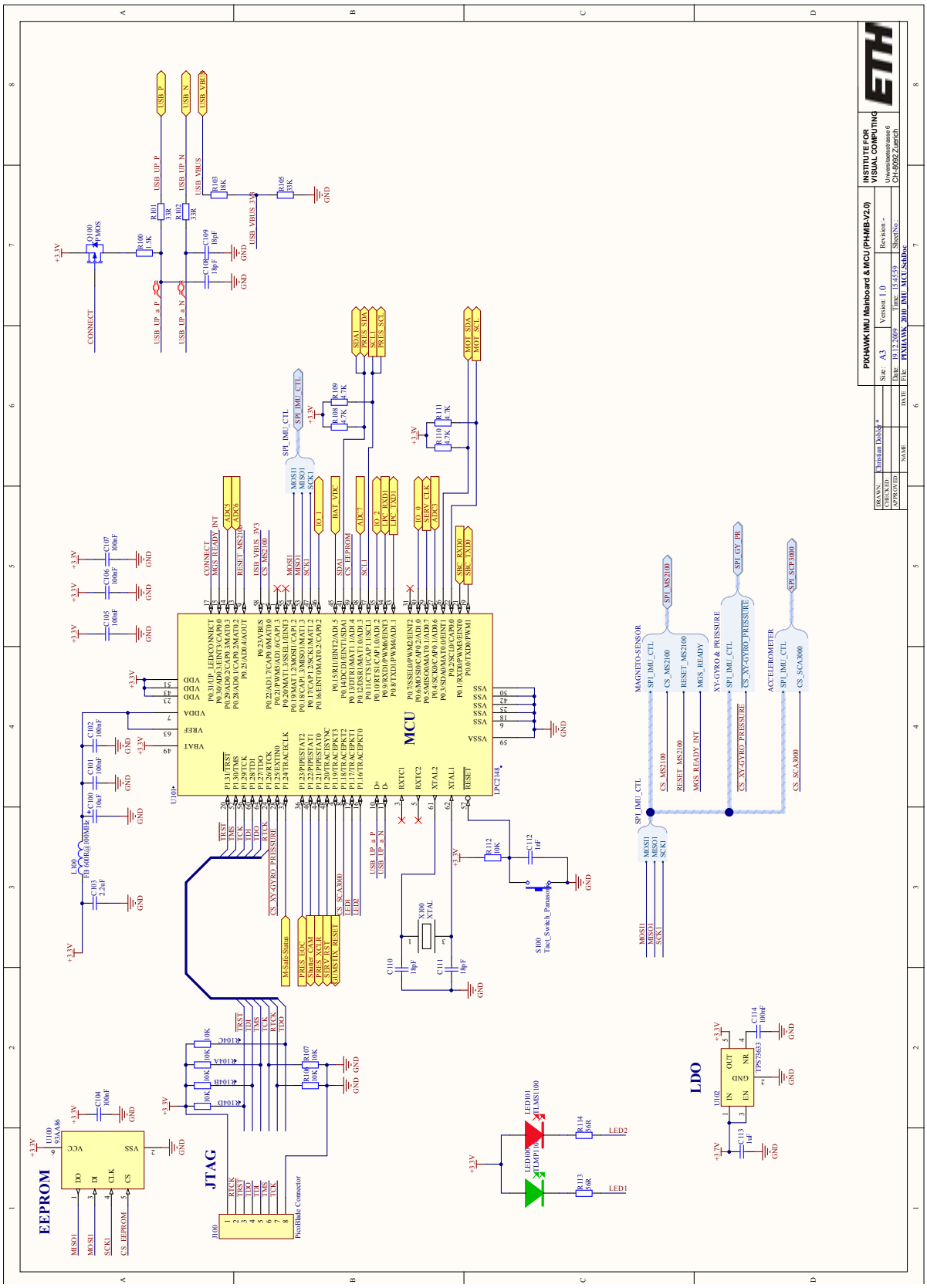


Figure 2.13.: MCU sheet of the schematics of the IMU prototype

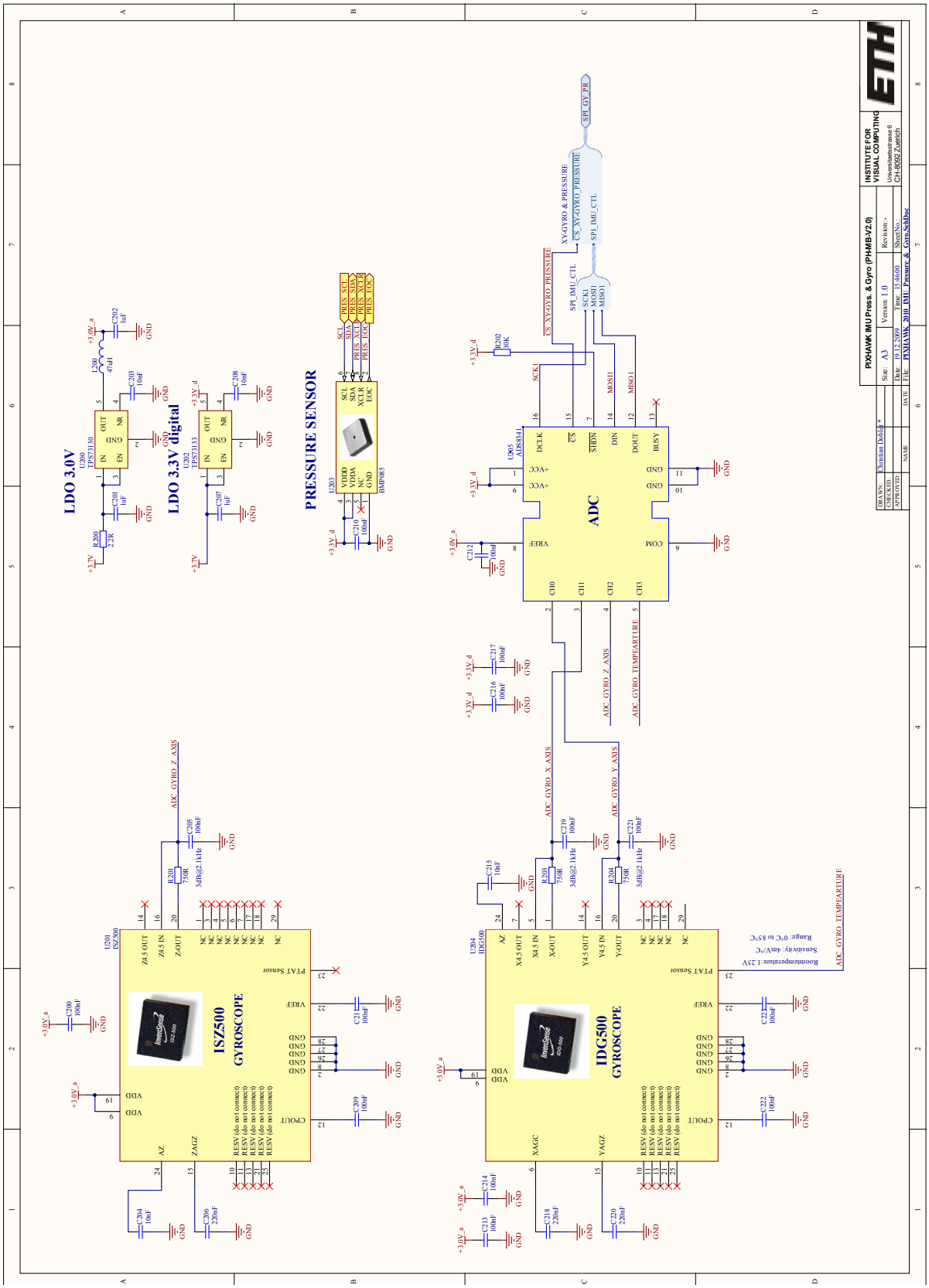


Figure 2.14.: Gyroscope and pressure sheet of the schematics of the IMU prototype

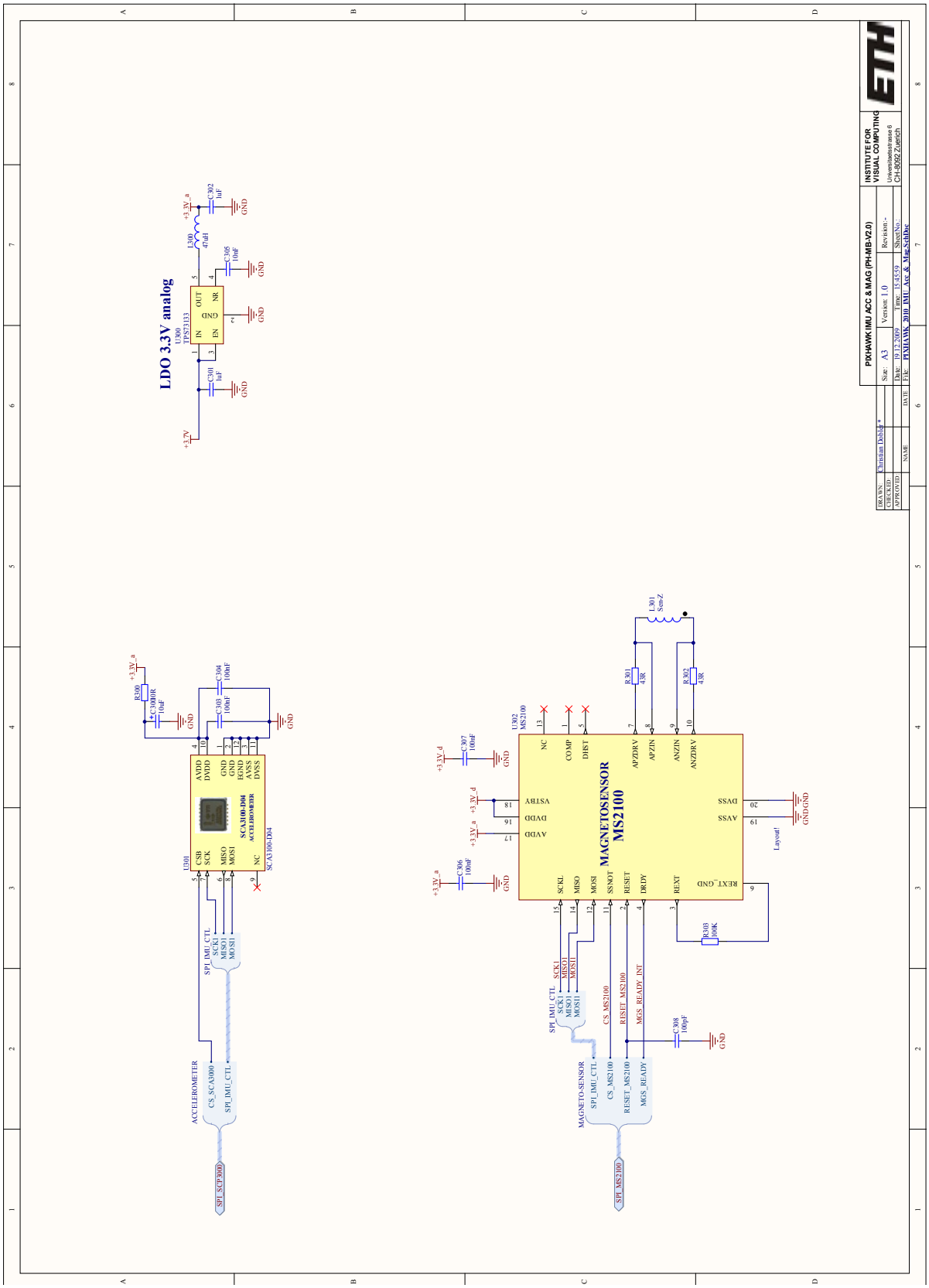


Figure 2.15.: Accelerometer and magnetometer sheet of the schematics of the IMU prototype

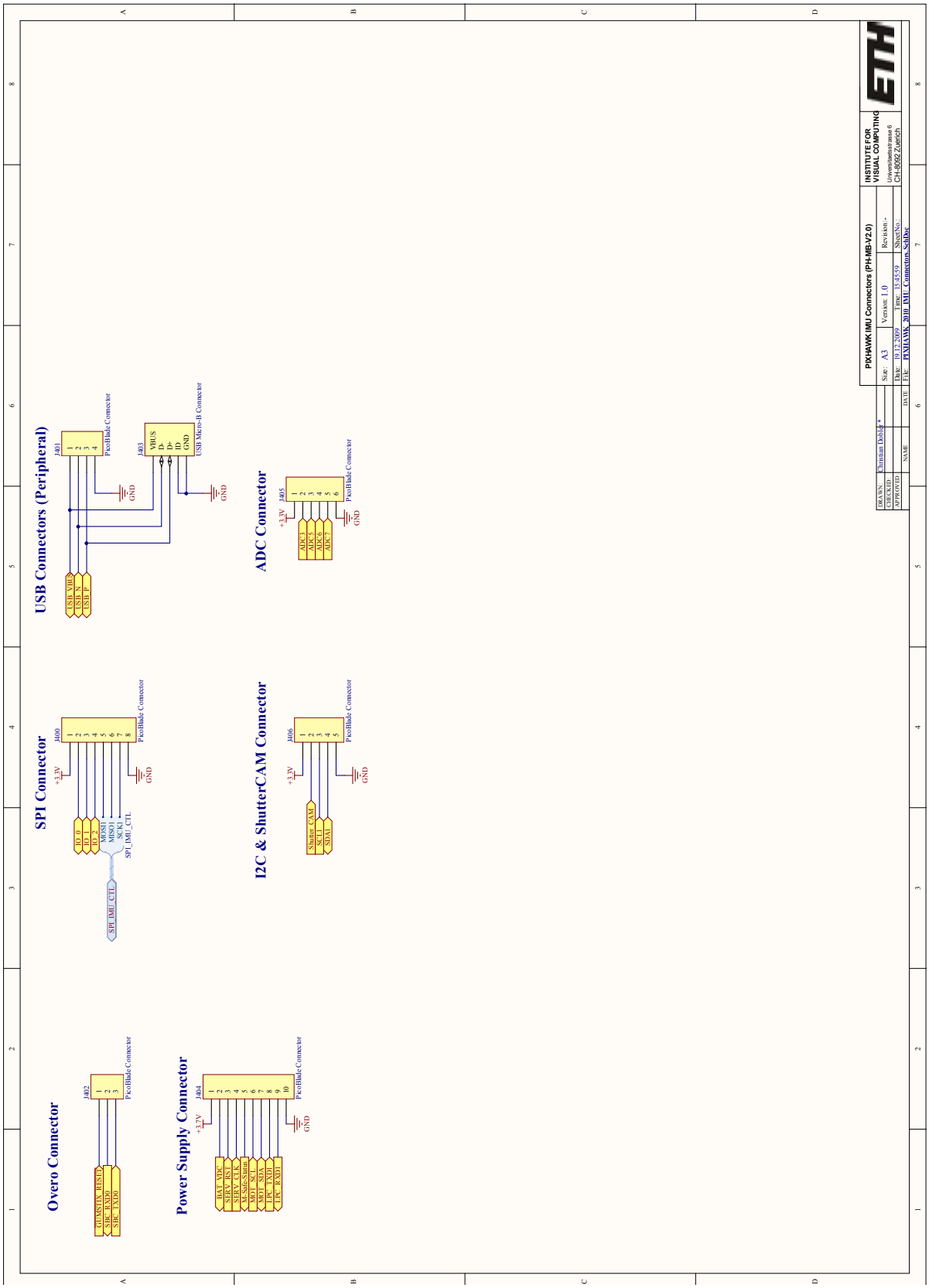


Figure 2.16.: Connectors sheet of the schematics of the IMU prototype

2.3. Baseboard for Gumstix Overo SBC

2.3.1. Overview

The *Gumstix Overo Fire [31]* is the Single Board Computer (SBC) used on the coaxial PIXHAWK helicopter model. It comprises a complete computer able to run an OpenEmbedded Linux distribution on a tiny PCB. The main specifications of the **Gumstix Overo Fire SBC** are as follows:

- **System-on-Chip (SoC): TI OMAP 3530 [32]**
  - 600 MHz ARM Cortex-A8 with NEON SIMD Coprocessor
  - 430 MHz TMS320C64x+ DSP (fixed point, six parallel vector units)
  - 110 MHz SGX GPU
- **RAM:** 256MB mDDR (2.7 GByte/s)
- **Flash:** 256MB NAND
- **Wireless Connections:** Wi2wi W2CBW003 System-in-Package
  - Wifi 802.11b/g
  - Bluetooth 2.0 EDR
- **Interfaces:** 2x USB 2.0, SPI, I2C, PWM, ADC, 2x UART
- **Storage Solution:** on-board microSD/SDHC card slot (tested with 8 GB)
- **Size:** 17mm x 58mm x 4.2mm
- **Weight:** 4g
- **Complete Specifications:** http://pixhawk.ethz.ch/electronics/overo

The goal of the new design was to develop a baseboard for a single Overo Fire SBC making all the different digital interfaces (i.e USB, UART, SPI and I2C) provided by the SBC available to external peripherals. Furthermore the number of USB Interfaces needed to be increased. This was necessary because more than one USB camera will have to be used on the MAV in the future due to the development of new computer vision algorithms, e.g. stereo vision. Additionally one USB interface should be used to provide more UART interfaces by means of a USB-to-UART converter. This will offer the possibility to attach external UART peripherals like GPS or wireless communication[2] modules.

Another requirement was the possibility to connect multiple Overo Baseboards together to form a multiprocessor computing architecture. Currently the multiprocessor software subsystem is too unstable to use it. That is the reason why we only want to integrate one Overo SBC on the coaxial helicopter for the standard use case. But in the future multiprocessing might become an option and the hardware should be able to support it.

[2] This is necessary since the embedded wireless module on the SBC had not worked well in previous tests of the Overo Fire SBC.

2.3.2. Development

The architecture of the new Overo SBC electronics is shown in figure 2.17. The Gumstix Overo SBC is connected to the PCB via two 70 pin AVX 5602-14 connectors. Since the Overo SBC uses 1.8V logic levels, bidirectional voltage level translators ("level shifters") have to be deployed on all digital connections (except USB) to provide standard 3.3V logic to external components. The logic level of the UART1 and the SPI signals is translated by the 8-bit bidirectional level shifter ADG3308 [33] from Analog Devices. The UART3 signal, which is connected to the Linux console, is available on the other AVX 5602-14 connector and is therefore level shifted by a separate 2-bit bidirectional level translator ST2329A [34] from ST Microelectronics. The logic level of the I^2C signals needs to be translated by a dedicated I^2C level shifter like the PCA9306DCUR [35] from Texas Instruments.

To increase the number of available USB ports a USB hub is connected to the High-Speed USB 2.0 host interface of the Overo SBC. The second USB interface provided by the Overo SBC, called USB OTG, is a USB On-the-Go [36] port, which means that this USB interface can take on the role of a host or a slave port depending on the device attached to it. If the USB hub was attached to this interface the On-the-Go functionality would be lost. Therefore the USB hub had to be attached to the standard USB 2.0 interface.

Three different 4-port USB 2.0 hub chips have been evaluated. Table 2.8 shows a comparison of the main characteristics of the evaluated hub chips and lists the advantages and draw-backs of each chip.

	μPD720114 [37]	**USB2514 [38]**	**ISP1520 [39]**
Manufacturer	NEC	SMSC	ST-Ericsson
Downstream Ports	4x USB 2.0	4x USB 2.0	4x USB 2.0
Package	48-pin TQFP	36-pin QFN	64-pin LQFP
Size	9x9x1.2 mm	6x6x0.5 mm	12x12x1.6 mm
Typ. Current Consumption (4 active dev.)	149 mA	215 mA	288 mA
Advantages	small package	smallest package	good availability
	lowest power consumption	low power consumption	good documentation
	best documentation		
Disadvantages	only available at Mouser at Mouser	lead-less package with GND pin under the package (not manually solderable)	largest package
			highest power consumption

Table 2.8.: Comparison of USB 2.0 hub chips

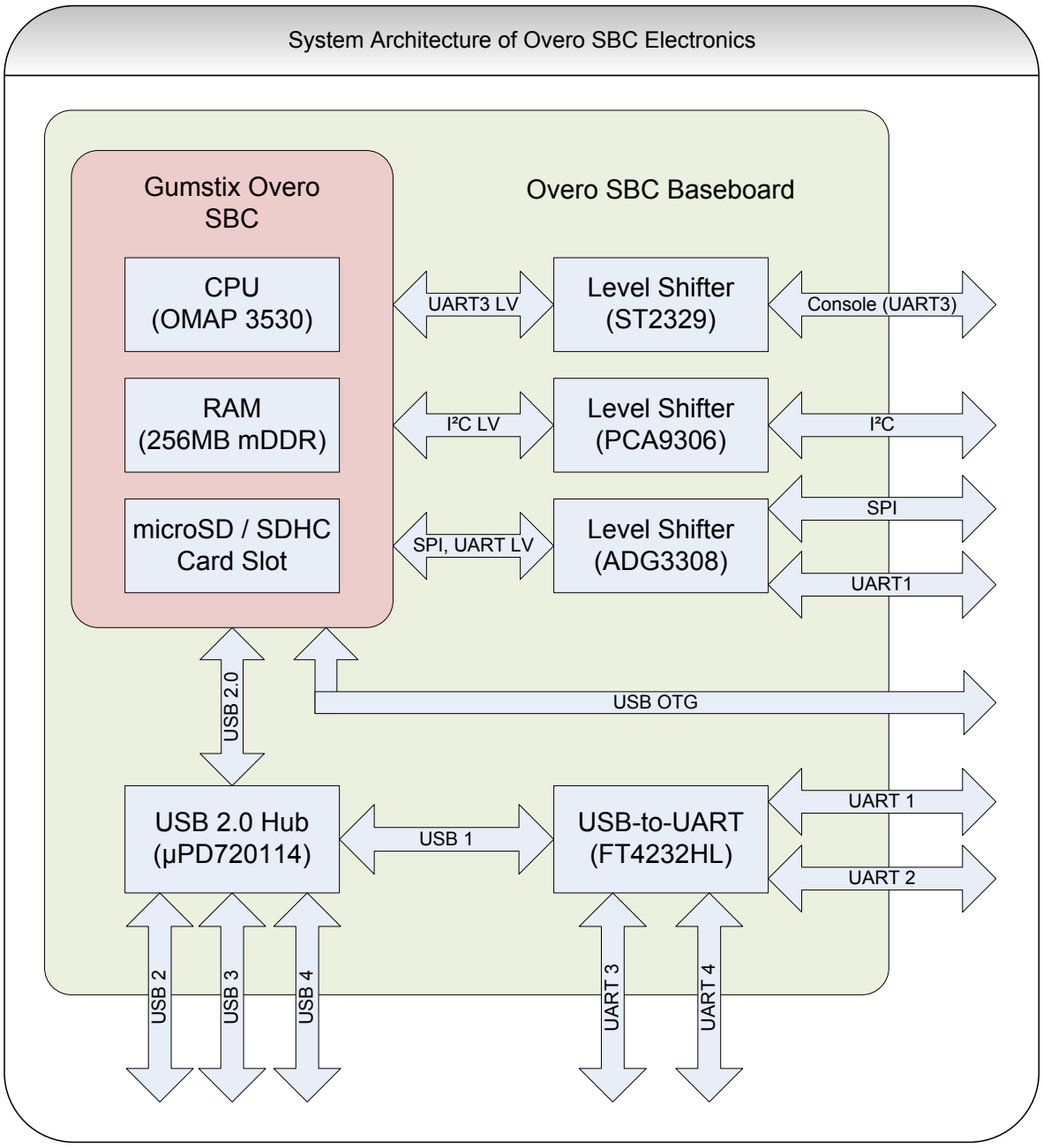


Figure 2.17.: System architecture of Overo SBC electronics

The NEC μPD720114 is the clear winner of this comparison, as it has the lowest power consumption and the 48-pin TQFP is a good compromise between size and solderability. Therefore this USB hub chip has been selected for use on the Overo baseboard.

A FT4232HL Quad-UART USB-to-UART converter [40] from FTDI, a well-known manufacturer of UART converters, has been connected to the first USB port of the USB hub chip to increase the number of available UART interfaces. The FT4232HL has been FTDI's top performance product at the time of selection. It provides four UART serial interfaces connected via a High-Speed USB interface to the USB host.

The Overo baseboards can be connected together for multiprocessing via one of the USB host ports on the first board and the USB OTG port configured as a slave on the second board to form an arbitrarily long multiprocessor ring. The only limiting factor is power, but 5 boards can be connected at minimum.

Figures 2.19 and 2.20 show the complete schematics of the prototype baseboard for the Overo SBC.

2.3.3. Manufacturing

Two prototypes of the PIXHAWK Overo Baseboard have been manufactured by Multi PCB in Germany. Figure 2.18 shows the top side of the baseboard PCB containing the connectors for the Gumstix Overo SBC.

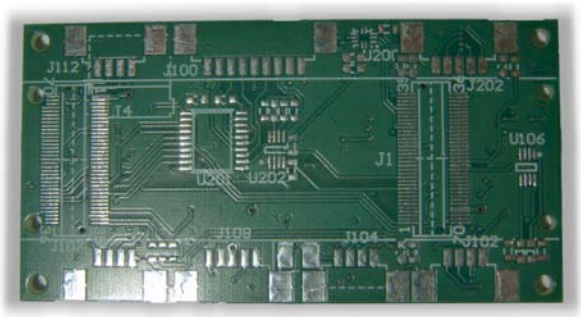


Figure 2.18.: Top side of Overo baseboard PCB

The components have been soldered manually as functional groups and electrically tested step-by-step. These electrical tests have exhibited unwanted short-circuits between a variety of pins of the NEC μPD720114 USB hub chip. As the first action the PCB was electrically tested for routing errors. No errors could be found in this test. Then a new chip was mounted pin-by-pin exhibiting the same short-circuits as soon as pin 24 (VDD25) was connected. This solder joint was then checked under a microscope but no bridge to any other pad could be detected. For this reason the pins of an absolutely new chip have been electrically tested on the desk without any connections. This test showed the unexpected result that the pins 24 (VDD25), 25 (USB Data-), 33 (VDD25) and 36 (VSS) are internally connected with an impedance as low as 6Ω. Also pins 12 (VDD25) and 42 (VSS) are internally connected. On

the other hand, pins that belong to the same net, e.g. VSS, are not connected internally. The same internal connections could be measured on all of our NEC USB hub chips.

This issue has been reported to the manufacturers (NEC) support department by email. Sadly, the manufacturer has never responded to this support inquiry. To exclude the possibility that all of our hub chips got broken during transport another five of these ICs were ordered from our supplier. But all of these chips showed the same defects. As a last action the USB hub chip and all of its necessary external components were mounted. Only the power supply components were also mounted on the Overo baseboard. Then the board was connected to a laboratory power supply and powered with the current limited to 0.5A. This test was meant to show whether the detected short-circuits only exist when power is off or if they are permanent. Since the power supply showed the maximum allowed current output of 0.5A and the voltage dropped immediately to 1.2V (from 3.7V nominal output) during this test, it can be deducted that the internal short-circuits are permanent. This means that either the chip has permanent defects or the pin map in the datasheet is faulty. But we just don't know because the manufacturer did not respond to the support request.

For this reason the USB hub chip would need to be replaced to use the full functionality of the Overo baseboard. This has been postponed to the absolute end of this master thesis as the Gumstix Overo SBC has been superseded in the meantime by the new much more powerful COM Express SBC. This will be described in detail in section 2.4. The Gumstix Overo SBC will most probably not be used any more by the PIXHAWK team in the upcoming MAV competitions. It is still interesting as a research platform though. Therefore it has been revised during the short time frame left at the end of this master thesis as described in section 5.3.

2.3.4. Specifications

- **Power Input:** 3.7V DC and 5.0V DC
- **Power Consumption:** max. 400mA @ 3.7V $\approx$ 1.5W
- **Single Board Computer:** Gumstix Overo Fire
- **External Interfaces:**
  - 3x USB Micro-A and 3x USB on PicoBlade (2 shared)
  - 1x USB Micro-AB and USB on PicoBlade with USB On-the-Go functionality
  - 1x SPI with 2 chip select lines (shared connector with I^2C)
  - 1x I^2C (shared connector with I^2C)
  - 4x UART interface (available)
  - 1x UART interface for communication with IMU
  - 1x UART interface connected with console
  - 2x LED on GPIO pin
- **Size of PCB:** 58 x 30.2 mm
- **Weight (completely assembled):** 12g (with Overo SBC mounted: 16g)

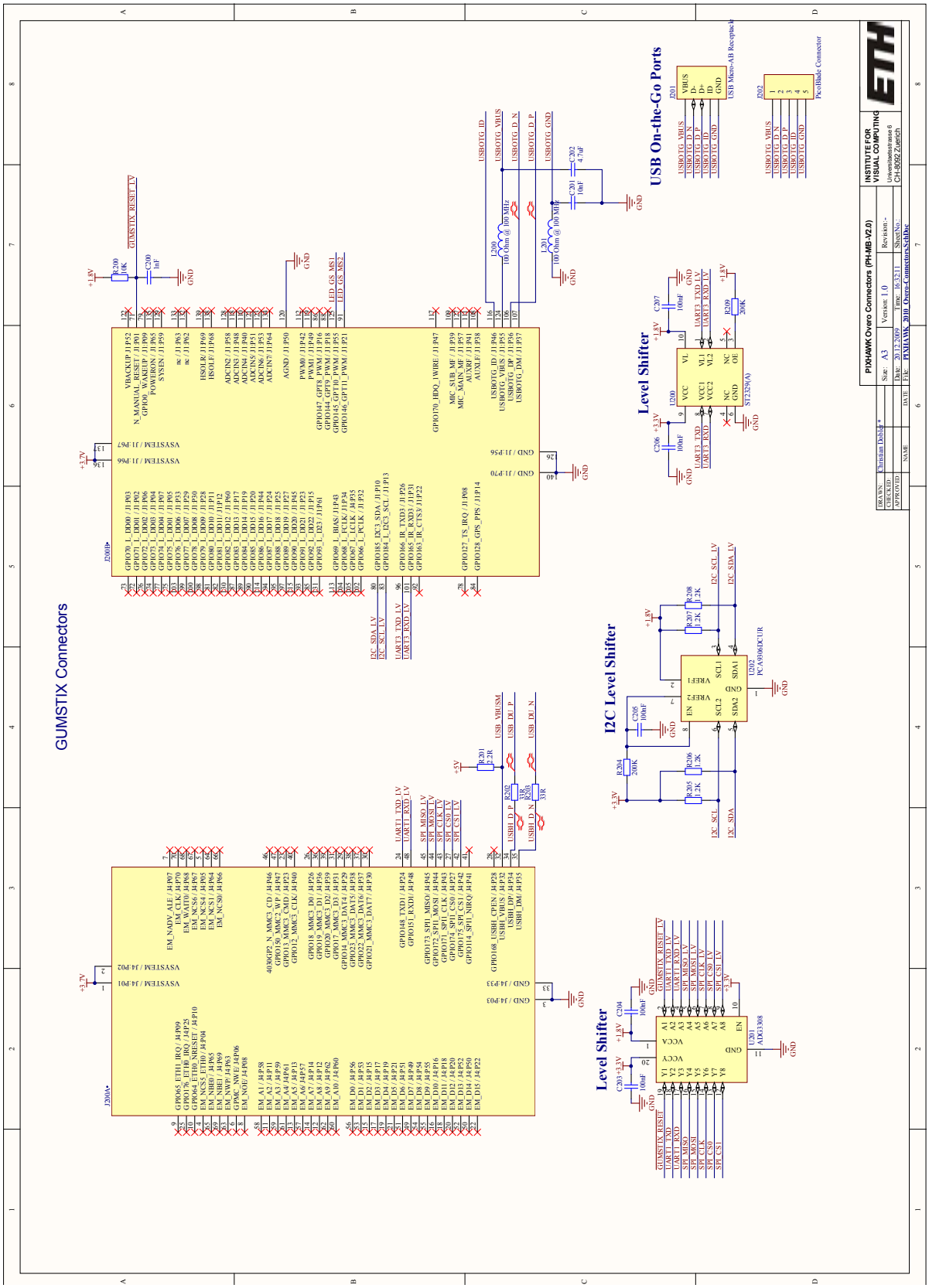


Figure 2.19.: Schematics sheet of Overo sockets and logic level translators

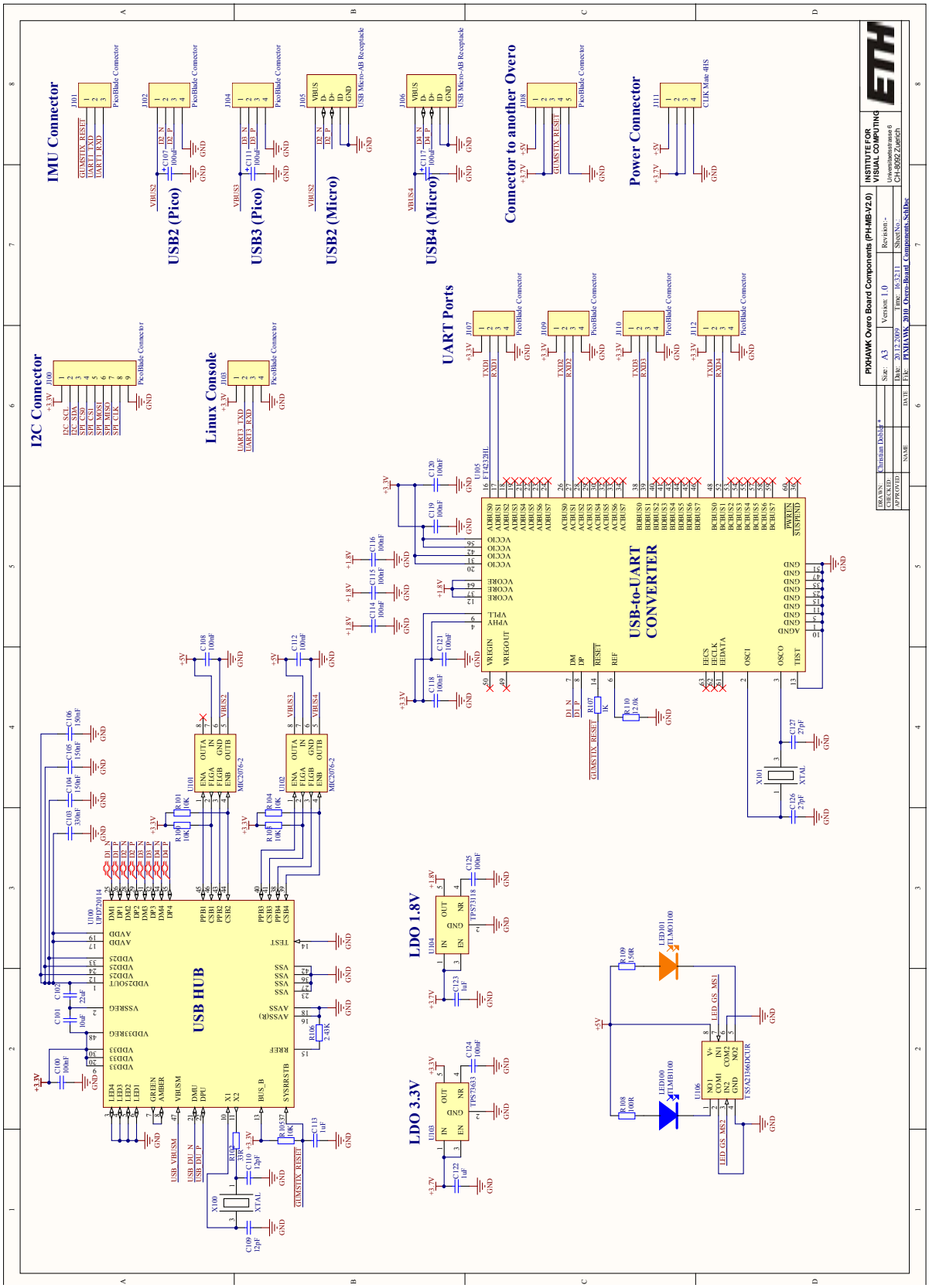


Figure 2.20.: Schematics sheet of main components on the Overo baseboard

2.4. Baseboard for COM Express SBC

2.4.1. Overview

Ongoing research on vision algorithms by other PIXHAWK team members has shown that the Gumstix Overo SBC does not provide enough processing power to run vision algorithms that allow full autonomy of the MAV. Currently vision algorithms for stereo vision, visual SLAM and position hold are being implemented for the PIXHAWK platform. To run these algorithms in real time, which would be necessary for full autonomy of the MAV, the on-board SBC electronics have to provide the processing power of current notebook systems. Therefore new single board computer architectures comprising current notebook CPUs had to be evaluated.

2.4.1.1. SBC Form Factor Selection

The most important factor regarding the selection of a new SBC is size. Since PIXHAWK participates in Micro Air Vehicle competitions, the new SBC is not allowed to increase the overall size of the MAV by too much. Weight is also a factor, but its importance can be reduced by a new mechanical design of the MAV, which would be necessary for the new SBC electronics in any case. The coaxial helicopter design will be replaced by a quadrotor design that can lift more weight and is much better controllable. This makes the form factor the deciding aspect for the selection of a new SBC.

Table 2.9 lists the different standardized "Computer on Module (COM)" form factors that have been evaluated. It also names the CPUs available for these form factors (at the time of evaluation). It is important to use a standard COM form factor as this guarantees a wide range of available products. Also new COMs can be expected to be released when new CPUs become available that can be easily integrated into our electronics. Therefore the PIXHAWK team will be able to benefit from the progress in CPU development in the future. On top these standardized form factors comprise standard x86 compatible CPUs, which means that they work just like any "Personal Computer" and standard operating systems can be used without any adaption.

The smallest form factor evaluated is *mobile-ITX* developed by VIA Technologies. The available mobile-ITX COMs use VIA's C7-M CPU, which is a low cost, ultra low voltage processor. The performance of the VIA C7-M is by far lower than the performance of current notebook processors. As section 4.3.1 shows the C7-M is even considerably slower than the Intel Atom. This fact removes the VIA ITX COMs from the selection.

ESmini is a miniaturized derivative of the ESMexpress form factor developed by the VITA standards organization. The available ESmini COMs provide Intel Atom CPUs as the top performance option. The Intel Atom is also significantly slower that current notebook (not netbook) processors. In section 4.3.1 a benchmark comparison between an Intel Atom and an Intel Core 2 Duo processor will be shown to assess the difference in performance in more detail. The restriction to Intel Atom CPU is a strong disadvantage of the ESmini COMs compared to the only slightly larger form factors comprising Intel Core 2 Duo CPUs.

Form Factor	PCB Size	Available CPUs
mobile-ITX [41]	60 x 60 mm	VIA C7-M
ESMini [42]	95 x 55 mm	Intel Atom
PC/104(-Plus) [43]	96 x 90 mm	Intel Core 2 Duo Intel Pentium M ULV AMD Geode
Pico-ITX [44]	100 x 72 mm	VIA C7-M VIA Eden ULV
COM Express [45] (compact)	95 x 95 mm	Intel Core 2 Duo Intel Atom
ETX/XTX [46]	114 x 95 mm	Intel Core 2 Duo Intel Atom

Table 2.9.: Comparison of standardized COM form factors

The PC/104(-Plus), COM Express (compact) and the *ETX/XTX* form factors are close in terms of size and performance. But the PC/104(-Plus) is actually more of an embedded computer than a computer on module (COM), i.e. it already provides peripheral connectors, e.g. USB, SATA, RJ-45, etc., on board. This is not exactly the idea of a COM. A COM should provide all the interfaces of the computing platform to the user on an aggregate connector. The user can than decide which signals should be made available to peripherals by designing a custom baseboard. This is also what we wanted to do and therefore the PC/104(-Plus) form factor was deselected. The *COM Express (compact)* form factor is slightly smaller than ETX/XTX and has therefore been chosen as the form factor of our new SBC electronics.

COMs adhering to the COM Express form factor can also be nicely integrated into the mechanical frame of a quadrocopter due to their quadratic outline. The mechanical frame for the new PIXHAWK quadrotor MAV has been designed by Philippe Petit in parallel to the development of the SBC baseboard and is shown in figure 2.21. It can clearly be seen that the new flight electronics will be fitted perfectly in the center of the quadrocopter providing the best possible balance for the MAV.

2.4.1.2. Evaluation of COM Express Modules

Now that the form factor had been selected, actual modules complying to the COM Express standard had to be evaluated. Table 2.10 shows two COM Express Modules comprising an Intel Core 2 Duo CPU with their main specifications.

The MSC COM offers the slightly faster SP9300 CPU compared to the SL9400 that is integrated on the Kontron COM. But the Kontron COM makes up for this performance loss with DDR3 memory instead of DDR2 memory as on the MSC COM. This means that these COMs are equal performance wise and that performance is not the decisive aspect.

The important difference between these two modules is that the Kontron COM offers a wide range power supply from 8.5V to 18V while the MSC COM needs a regulated 12V power

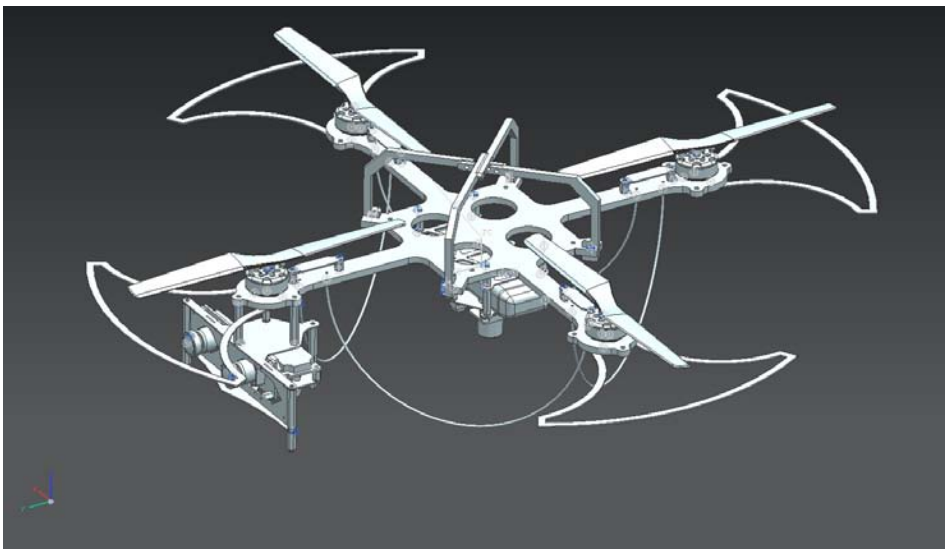


Figure 2.21.: CAD model of the mechanical frame of the new PIXHAWK quadrocopter [1]

	microETXexpress-PC [?]	**CXC-GS45 [47]**
Manufacturer	Kontron	MSC Vertriebs GmbH
Chipset	Intel GS45	Intel GS45
CPU	up to Intel Core 2 Duo SL9400, 1.86 GHz	up to Intel Core 2 Duo SP9300, 2.26GHz
GPU	Intel GMA X4500MHD	Intel GMA X4500MHD
RAM	up to 4 GB DDR3 SO-DIMM (1066 MHz)	up to 4 GB DDR2 SO-DIMM (800MHz)
Power Input	8.5V - 18V DC	12V ±10% DC

Table 2.10.: Comparison of two different COM Express modules comprising an Intel Core 2 Duo CPU

input. This is a great advantage of the Kontron COM because the PIXHAWK team uses 11.1V brushless motors requiring an 11.1V lithium polymer battery to be mounted on the MAV. This means that MSC COM would always be operated at the margin of its allowed power input possibly causing unexpected power losses. Even more of a problem is that these lithium polymer batteries keep their nominal voltage until they are nearly empty. Then the output voltage drops quickly to as low as 9V. This could cause the problem that the MSC COM will permanently loose power before the low battery detection alerts the operator of the MAV since the COM is always operated at the margin of the allowable power input. The Kontron COM will not suffer from these issues as the battery voltage never falls below its minimum input voltage of 8.5V.

For this reason the Kontron microETXexpress-PC COM Express module has been selected for the new PIXHAWK SBC electronics. This module is shown in figure 2.22.

Figure 2.22.: Kontron microETXexpress-PC COM Express module [5]

2.4.2. Development

The custom baseboard for the COM Express module was designed to comply with the COM Express standard by following the *PICMG COM Express Carrier Board Design Guide [48]*. The most crucial part of the baseboard design was the *power supply*. The reason for this is that the COM Express imposes tight constraints on the power supply implementation. The two most important constraints are the slew rate limit to the input voltage rise according to the ATX specification and that the power input has to be regulated to 12V with a maximum deviation of ±5%. If we conected the 11.1V lithium polymer battery mounted on the MAV directly to the power supply pins of the COM, we would violate both constraints. Also the 12V regulated voltage input requirement can not be met with a 11.1V battery at all. But on the other hand the COM Express module selected for the PIXHAWK MAV supports much lower input voltages. Therefore an approach with two distinct power supply rails has been chosen, where one rail completely complies to the COM Express standard and the other complies to the slew rate constraint but not to the input regulation.

Figure 2.23 depicts the schematics of the 12V power supply to ease the following description of its functionality. Any battery with a rated voltage between 14.5V and 36V can be attached to battery pads labeled with "18.5V" (X200, X201). This battery supply voltage is fed to a PTN78020W [49] DC/DC converter module (U200) from Texas instruments. This DC/DC converter generates a 12V output voltage regulated within ±3% over the complete operational range when the adjustment resistor (R201) is set to 732Ω. The PTN78020W has a maximum output current of 6A at 12V (= 72W), which is more than required by the standard (50W). This power rail also meets both constraints of the COM Express standard, i.e. any COM Express module can be safely powered from this rail. It can be selected by

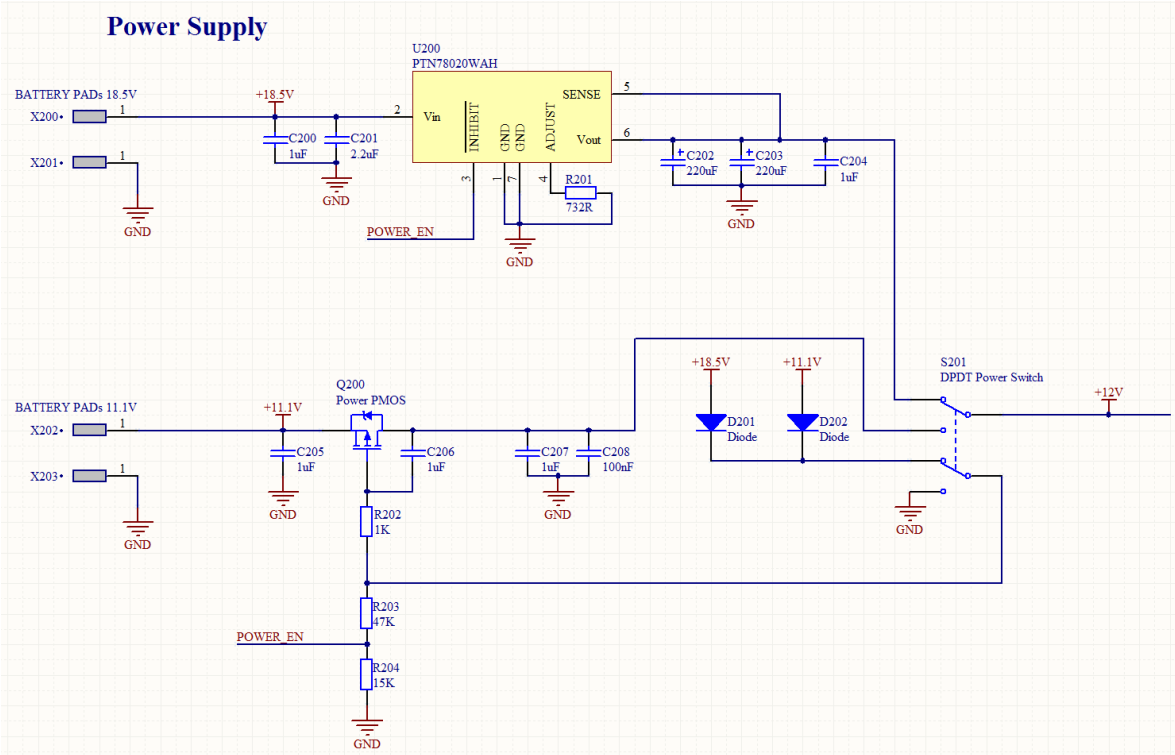


Figure 2.23.: Schematics of 12V power supply design

setting the actuator of the mechanical switch S201 to position 1. This action connects this rail with the 12V power rail of the COM and pulls the "INHIBIT" input of the PTN78020 high, which makes it raise its output slew-rate limited to 12V.

The battery pads labeled with "11.1V" (X202, X302) are only to be used with an 11.1V lithium polymer battery. The battery supply voltage is applied to the source of the slew-rate limited power PMOS transistor SUM110P04-05 [50] (Q200) from Vishay Siliconix. If the voltage applied to the gate of the PMOS transistor is >8V, the transistor is in blocking state, which happens if the mechanical switch (S201) is set to position 1. When the actuator of S201 is moved to position 2, the gate voltage is decreased smoothly via the slow-start circuit composed of R202 and C206. This leads to a slew-rate limited increase of the drain voltage which is now connected to the 12V power rail of the COM. This power rail can be used with any COM Express module that provides a wide power input range like the selected Kontron COM.

The 5V and the 3.7V power rails are generated from the 12V power supply rail in the same manner as described in section 2.1. Also all the other components integrated on the PIXHAWK power supply board like the servo controller, the motor connectors and the XBee [7] socket have been implemented on the COM Express baseboard. This means that only the IMU board has to be used as an external component with this SBC. A cutout in the PCB has been designed, where the IMU should fit in as it can be seen in figure 2.25. Figure 2.24 shows the system structure of the new COM Express SBC electronics while figures 2.26 and 2.27 depict the completely assembled COM Express baseboard with the main components

and interfaces labeled. The complete schematics of the prototype baseboard can be found in figures 2.28 and 2.29 at the and of this section.

2.4.3. Manufacturing

The PCB layout of the COM Express baseboard was finished just before Christmas 2009. Because our previously used prototype PCB manufacturer Multi PCB discontinued production over the Christmas holidays till January 4th 2010, a new manufacturer had to be found to get on with the development. Since assumably all European and American PCB manufacturers stop production over the Christmas holidays, Chinese manufacturers have been evaluated. The offer of the Chinese PCB manufacturer Gold Phoenix PCB was the most convincing. Therefore Gold Phoenix PCB was entrusted with the production of the COM Express baseboard PCB. Production time was short and the quality of the PCBs was the best of the manufacturers we have been using. The drill precision was perfect and the quality of the copper tracks and soler mask was great. The blank PCB is shown in figure 2.25.

The different building blocks of the COM Express baseboard have been manually soldered and electrically tested step-by-step. A completely assembled baseboard prototype with labels for the main building blocks and interfaces is shown in figures 2.26 and 2.27.

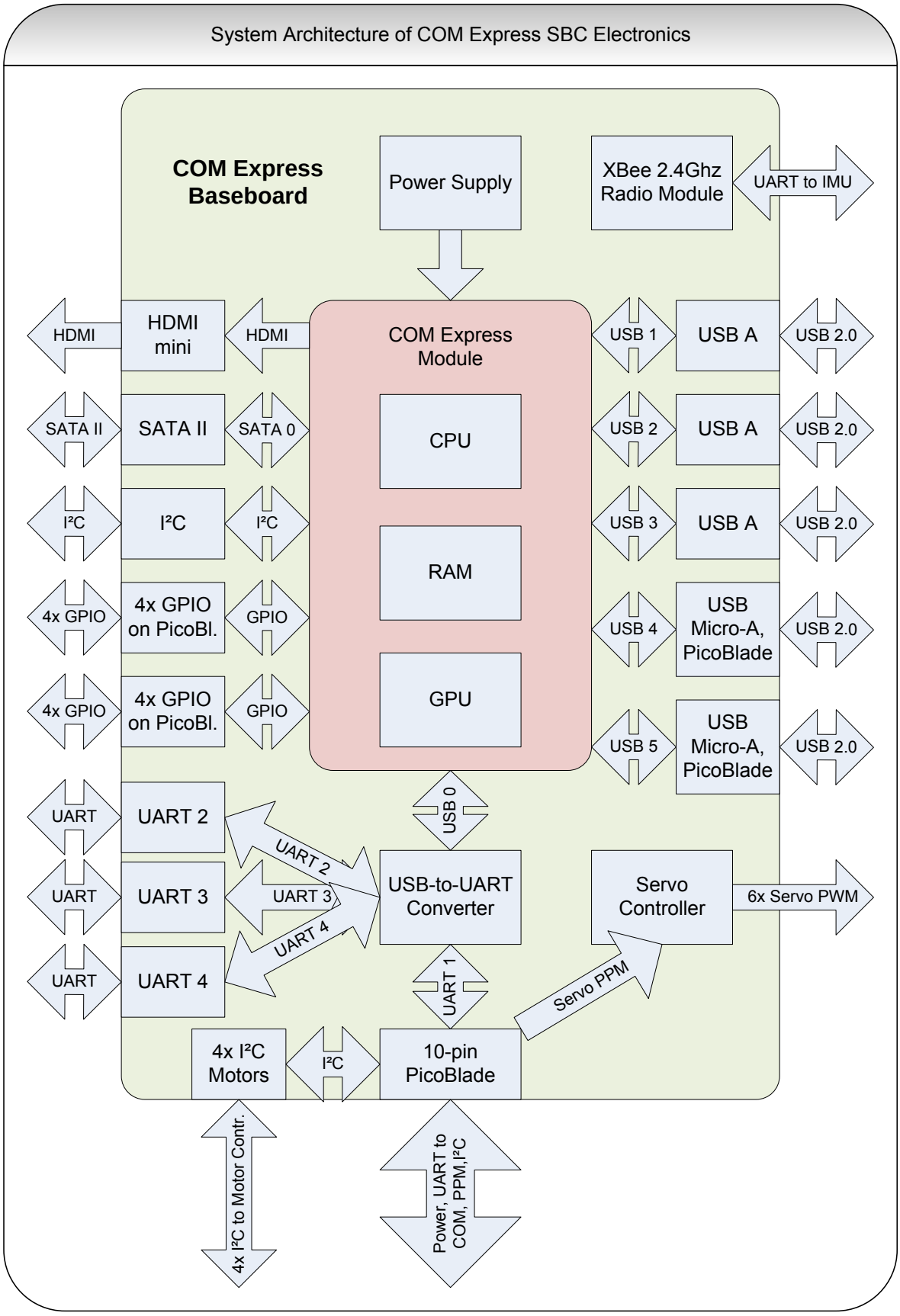


Figure 2.24.: System architecture of COM Express SBC electronics

Figure 2.25.: PCB of COM Express baseboard

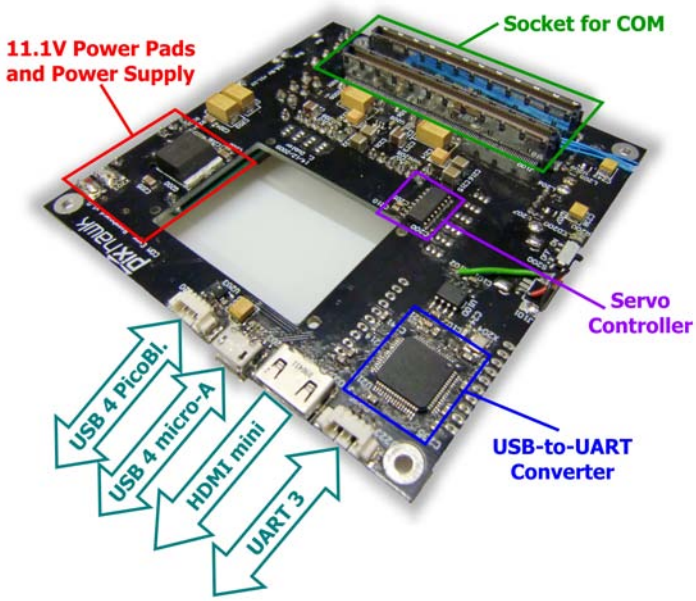


Figure 2.26.: Top side of assembled COM Express baseboard with main blocks labeled

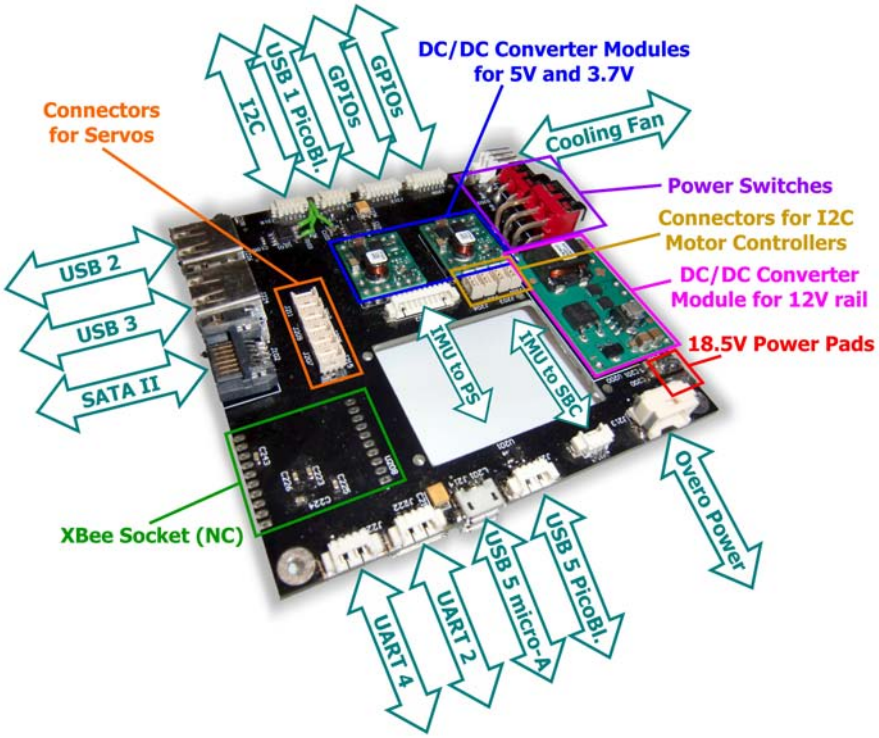


Figure 2.27.: Bottom side of assembled COM Express baseboard with main blocks labeled

2.4.4. Specifications

2.4.4.1. Kontron COM Express Module Specifications

- **CPU:** Intel Core 2 Duo SL9400, 1.86GHz, 6MB L2 Cache
- **RAM:** 2GB DDR3, 1066MHz
- **Chipset:** Intel GS45
- **GPU:** Intel GMA X4500MHD
  - 10 Vector processors
  - Programmable shaders / GLSL / CL support
- **Size of PCB:** 95 x 95 mm
- **Weight:** 89g (incl. RAM)
- **Kontron Part Number:** 36004-0000-19-2

2.4.4.2. COM Express Baseboard Specifications

- **On-Board Power Supply:**
  - **12V / 72W** regulated power output generated by a TI PTN78020WAH switching DC/DC converter module complying with COM Express standard (14.5V to 36V battery necessary)
  - direct, slew-rate limited 11.1V power input from battery as an option (works with Kontron COM Express module but does not comply with standard)
  - **5V rail:** 30W (generated by a TI PTH08T230W DC/DC converter module)
  - **3.7V rail:** 22W (generated by a TI PTH08T230W DC/DC converter module)
- **Power consumption under full load (CPU and GPU) without peripherals:**
  - **Peak:** 29.4W (2.6A @ 11.3V)
  - **Average:** 27.1W (2.4A @ 11.3V)
- **External Interfaces:**
  - 1x SATA-II receptacle with 3-pin PicoBlade power connector for SATA SSD
  - 1x HDMI mini (display)
  - 2x USB-A port
  - 3x USB on 4-pin PicoBlade shared with 2x USB micro-A ports
  - 2x GPIO connectors with 4 GPIO pins on each connector (8 GPIO connections in total)
  - 3x UART on 4-pin PicoBlade (emulated by a FTDI FT4232HL USB-to-UART converter [40] on USB port 0)
  - 1x I^2C on 4-pin PicoBlade
  - Interface to PIXHAWK Overo SBC module (including power)
  - Interface to PIXHAWK IMU module (including power)
  - Servo Controller for up to 6 servos
  - Motor Controller interface for up to 4 digital I^2C Motor Controllers
  - XBee Pro 2 Wireless Module for 2.4GHz RF communication
- **Size of PCB:** 95 x 95 mm
- **Weight:**
  - bare PCB: 29g
  - assembled: 81g

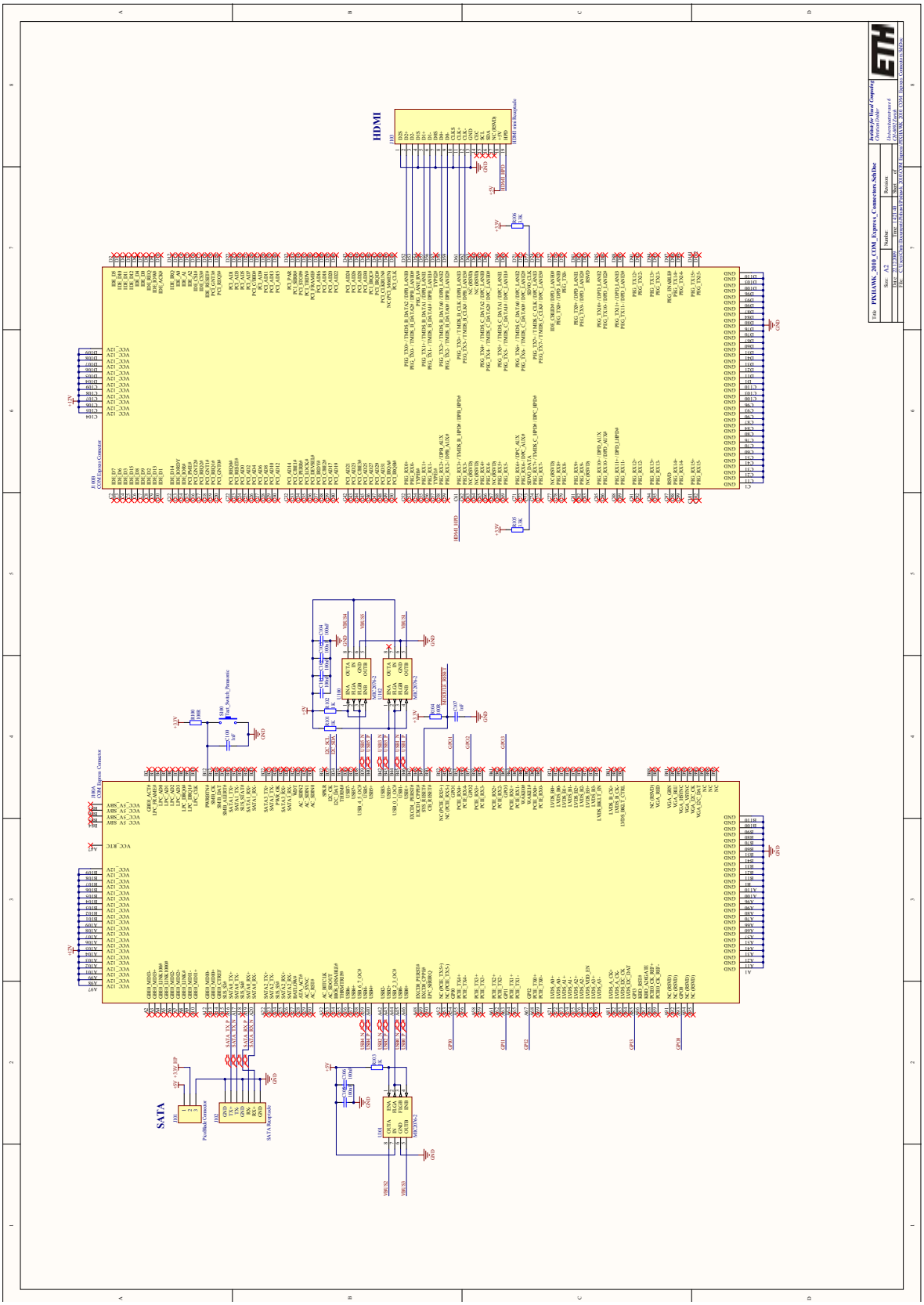


Figure 2.28.: Schematics sheet of COM Express Type 2 sockets

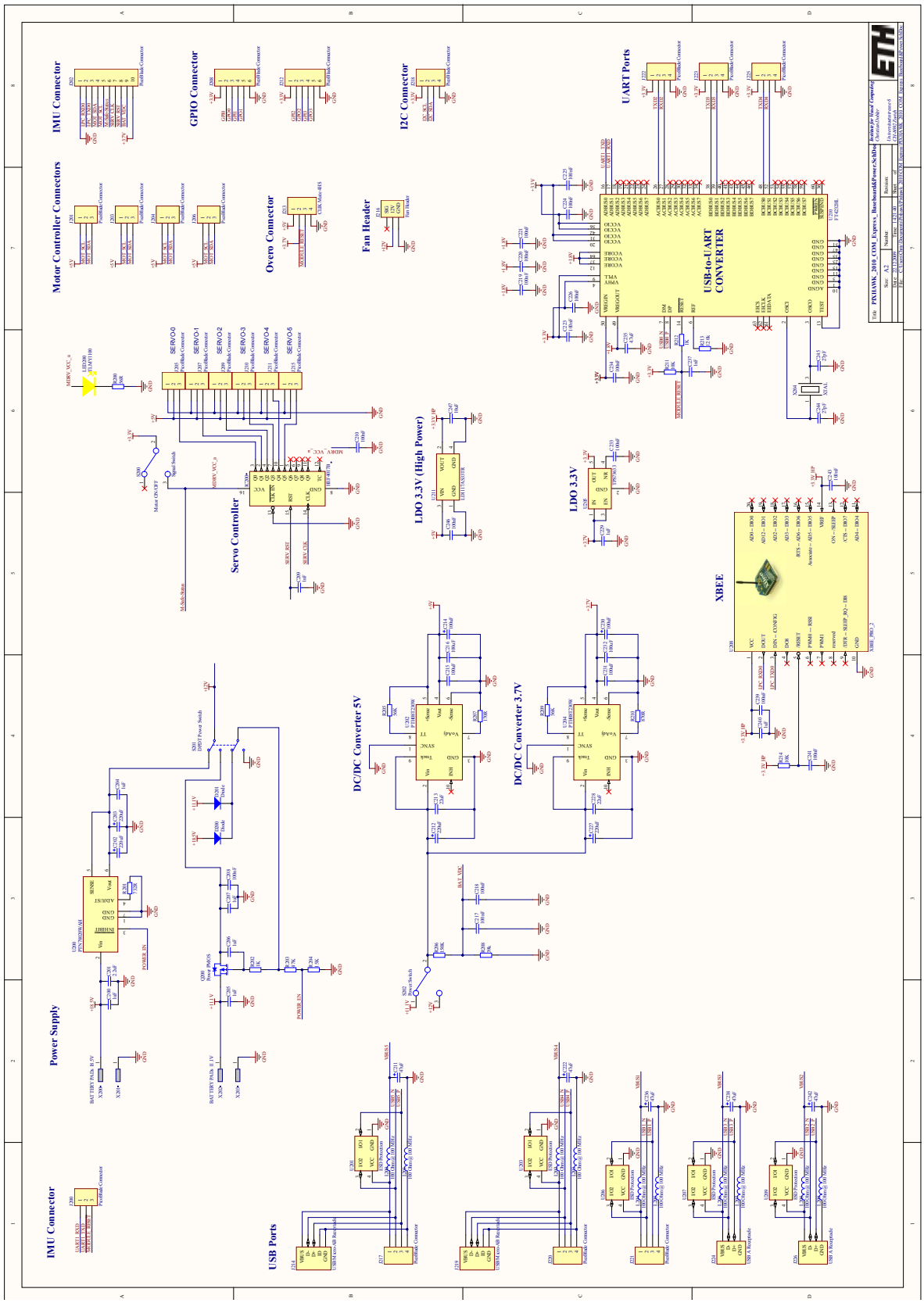


Figure 2.29.: Schematics sheet of main components on the COM Express baseboard

2.5. Bus-Powered USB-to-UART Adapter

2.5.1. Overview

Software testing for the vision system can now be done on any desktop PC because the new SBC uses the same instruction set and can run the same operating system. The only issue is, that standard PCs lack interfaces for embedded peripherals, such as UART, I^2C, etc. For this reason a small board has been developed that can be attached to one USB port of the host PC and provides 4 UART interfaces for the connection of embedded peripherals like GPS modules, digital sensors or RF modules. The USB bus voltage is used to provide the peripherals with a 3.3V supply voltage. No external power supply is necessary.

2.5.2. Development

The main component of this board is a FT4232HL Quad-UART USB-to-UART converter [40] from FTDI, that converts one USB port to 4 UART interfaces. The 3.3V power supply for the FT4232HL and the external UART peripherals is generated from the USB bus voltage by a TPS73633 [51] low dropout regulator from TI, that can source a maximum current of 400 mA. A minimum of 250mA is available to power the external UART devices. The complete schematics of this board is shown in figure 2.31 at the end of this section.

2.5.3. Manufacturing

Two prototypes of the bus-powered USB-to-UART adapter PCB have been manufactured by Multi PCB in Germany. The board has been assembled manually and electrically tested. Figure 2.30 shows the top side of the USB-to-UART adapter containing the FT4232HL Quad-UART USB-to-UART converter.

Figure 2.30.: 4-port USB-to-UART adapter module

2.5.4. Specifications

- **Power Input:** powered from USB bus
- **Power Output:** min. 250mA @ 3.3V
- **Interfaces:**
  - 1x micro-B USB 2.0 slave port
  - 4x UART with 3.3V power supply on 4-pin PicoBlade connector

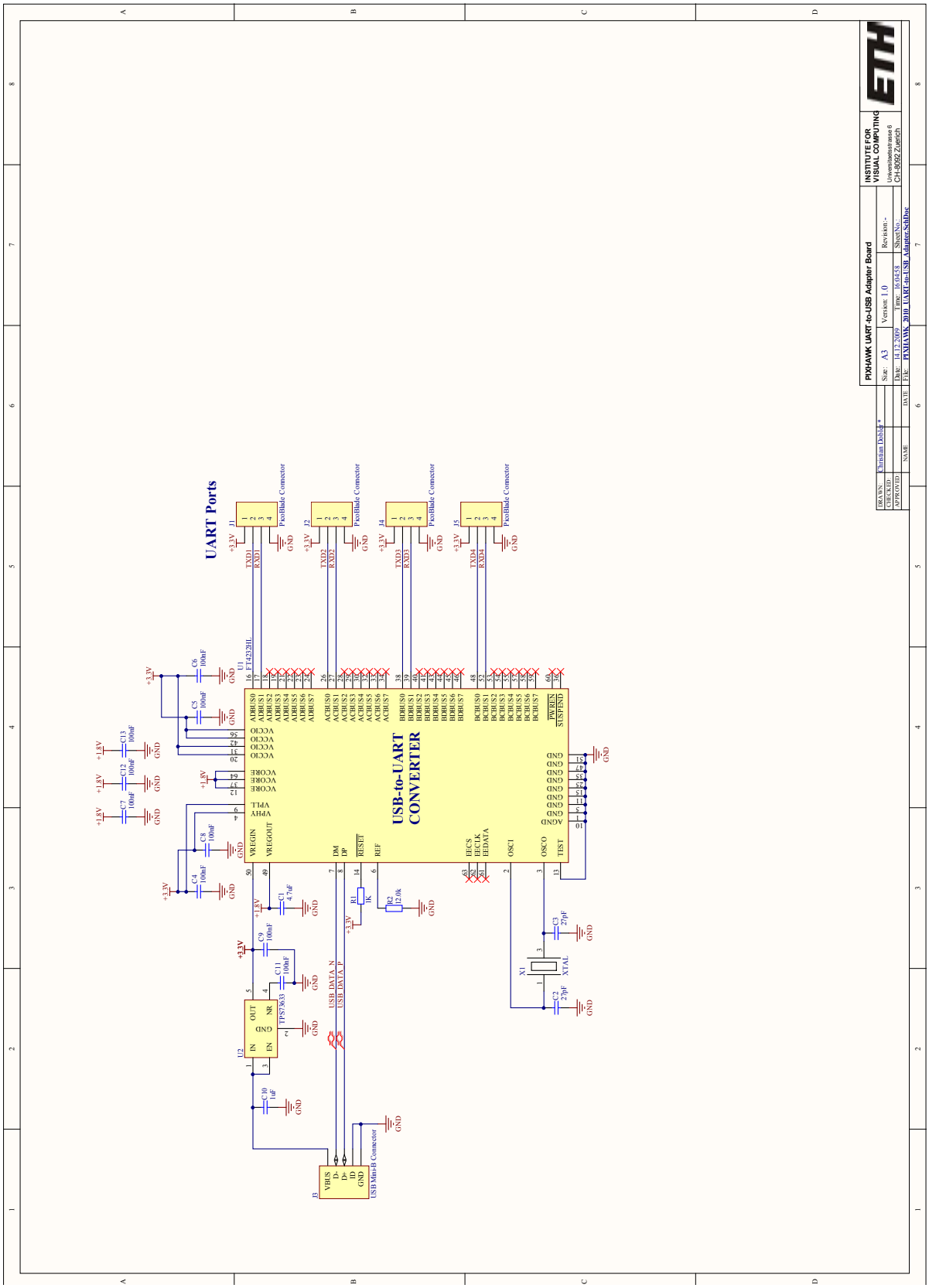


Figure 2.31.: Schematics of bus-powered USB-to-UART adapter

3. Embedded Software for IMU

This chapter deals with the embedded software running on the LPC2148 [15] ARM 7 based microcontroller integrated on the PIXHAWK IMU. This software is written in C code and compiled with the GNU `arm-elf-gcc` cross-compiler.

3.1. Codebase Selection

As a first task the codebase to be used as the basis for the embedded software running on the new PIXHAWK IMU had to be selected. Two options were evaluated:

1. **WASP codebase [52]:**
   WASP is an autopilot software project for MAVs based on the Paparazzi project [14] and refined by John Stowers from the CVG, ETH Zurich. It uses a sophisticated task scheduling architecture. But unfortunately it is only documented marginally. At the high level of abstraction used in this codebase the introduction to this software was judged to be hard with only fragments of documentation at hand. On top there was no Windows toolchain specified for the software development. This imposes additional setup overhead for system engineers using hardware development tools (such as Altium Designer), that are only available for the Windows operating system.

2. **PIXHAWK "flight code" codebase [53]:**
   This is the software used on the previous PIXHAWK flight electronics. Its main functionality is documented in prose in the PIXHAWK wiki and the API is described by a Doxygen documentation available on the PIXHAWK website. This facilitates the introduction to this codebase and allows for quick development successes. Additionally the existing drivers for the hardware parts that have not been exchanged in the redesign can be adopted without any changes. Also a complete Windows toolchain was promised to be available for the development. The draw-back of this software is its task scheduling, which is done by a time-driven main loop that is only executed every 20ms. The maximum execution rate (50Hz) is defined by the maximum refresh rate of the slowest component. This means, that because of one slow component the complete system has to be run at this speed.

The high complexity and the lack of documentation of the WASP codebase was rated to be too big an issue to achieve good progress within the time frame of this master thesis. Therefore the PIXHAWK codebase was selected as the basis of the new PIXHAWK flight code. The goal was to get all the on-board devices to run and to implement a different task scheduling solution which does not limit the execution rate of the main loop.

Therefore it was tried to set up the toolchain for the development used with the PIXHAWK codebase on a Windows PC. This failed at first. It turned out that the setup of this tool chain had only been done once before and that it had not worked stable. So the correct setup parameters had to be figured out to get this toolchain to run seamlessly. The correct setup process can now be found as a tutorial in the PIXHAWK Wiki.

3.2. Main Loop - Task Scheduling

The main loop consists of two parts: The initialization routine `main_init()` which is only executed once at the system startup. It initializes all the on-chip peripherals and external devices and sets up timers and interrupts. The second part is the `main_loop()` which contains an infinite while loop. In this while loop all the functionality is implemented that is necessary for orientation sensing and control of the MAV. The program code contained in `main.c` can be found in appendix C.1.

In the legacy PIXHAWK codebase the main loop has always been blocked at the beginning waiting for a periodic state generated by TIMER0. This periodic state event was triggered every 20ms (or any other selected period) defining a fixed execution rate of the main loop. It had been implemented like this to make sure that even the slowest sensor has finished its measurement during one program cycle. The draw-back of this approach is, that it also confines the sensor fusion, the control, the communication and faster sensors to the selected execution rate.

Therefore a new task scheduling scheme had to be developed that allows executing code parts at the rate best suited for each of them. For this purpose the new function `uint8_t ms_run_every(uint32_t ms, counter_id_t counter_id)` was added to main.c. The parameter `ms` is the period in milliseconds of a specific process. A specific counter that holds the value of the next execution time is associated with every `ms` value and is referenced by the second parameter `counter_id`. The allowed values for `counter_id` are the variables specified in the enum structure `counter_id_t` also implemented in `main.c`. To register a new period, i.e. calling `ms_run_every(uint32_t ms, counter_id_t counter_id)` with a not yet existing `ms` value, a new counter has to be registered in the enum structure `counter_id_t` and used in the function call.

The function `uint8_t ms_run_every(uint32_t ms, counter_id_t counter_id)` determines whether the system time has already reached the next execution time of the associated process referenced by `counter_id`. If no, the function just returns with the return value set to zero. If yes, it adds the value of `ms` to the current system time and stores the result as the next execution time for this `counter_id`. Then it returns with the return value set to one.

To schedule process execution at periodic time instants, these processes should be placed in the body of an if statement that calls `ms_run_every(uint32_t ms, counter_id_t counter_id)` to decide whether the code in its body is executed or not. The parameter `ms` should be set to the period length in milliseconds and `counter_id` to the associated counter.

Figure 3.1 shows the flow chart of how process scheduling is implemented now in the infinite while loop.

This way all sensors can be read out at their different maximum refresh rates. The function `sensors_read_out()` from `sensors.h` had to be split up into different functions for the different refresh rates. Appendix C.2 shows to new implementation of reading the sensors in `sensors.h`.

The communication receive function is looking for received information every cycle of the main loop, which runs at maximum speed, to make communication as real-time as possible. Data transmission is distributed on several functions that are executed at different periodic rates. The sensor fusion (complementary filter) and the control code have not been edited in the scope of this master thesis. Therefore they are still designed for a fixed main loop rate of 50Hz. This can be emulated in the new task scheduling scheme by just executing these functions every 20ms.

3.3. Drivers

This section will give a short overview of the device drivers developed for the new PIXHAWK IMU. The driver for the MS2100 [26] magnetosensor and the ADS8341 [25] ADC that converts the gyro measurements have been adopted from the legacy pixhawk codebase. Also the SPI subsystem did not have to be adapted. For completeness these drivers are also shown in appendix C.

3.3.1. I^{2}C Subsystem

Since with the BMP085 barometric pressure sensor an I^{2}C component had been integrated on the new IMU, a driver for the I^{2}C interface of the microcontroller had to be implemented. This I^{2}C interface driver can be used for the communication with any external I^{2}C device. Therefore it represents a subsystem to higher level device drivers. The I^{2}C subsystem is implemented in the files i2c.h and i2c.c that can be found in appendix C.3.

The I^{2}C subsystem API was designed to be as consistent as possible with the API of the SPI subsystem even though the hardware implementation of SPI and I^{2}C is completely different on the LPC2148 MCU. This way the developer of higher level device drivers will more easily understand the API of one subsystem if he / she already knows the other one.

The function `i2c_init()` has to be called once after system start-up to initialize the I^{2}C subsystem. This function has to be called before any higher level I^{2}C device is initialized.

The I^{2}C subsystem API defines a new datatype `i2c_package`. One `i2c_package` per I^{2}C transaction has to be constructed. Figure 3.2 depicts the different bytes of an `i2c_package`.

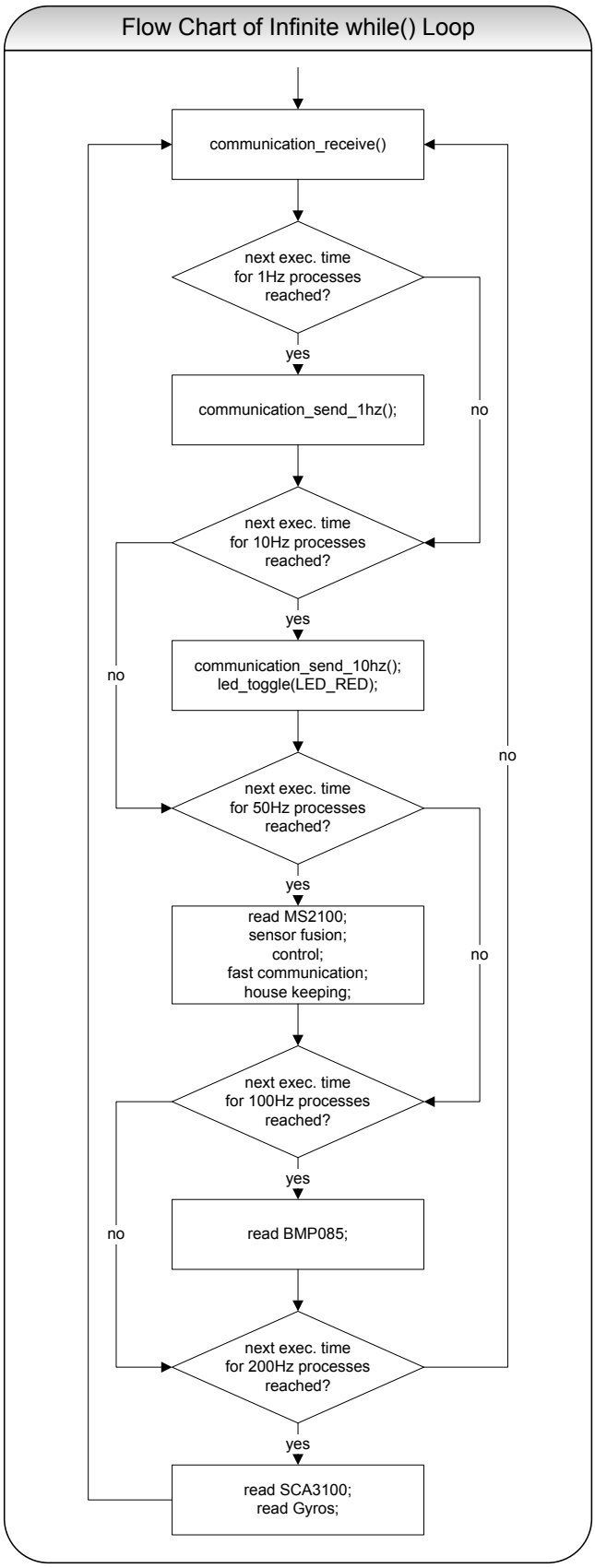


Figure 3.1.: Flow chart of infinite while() loop in main.c

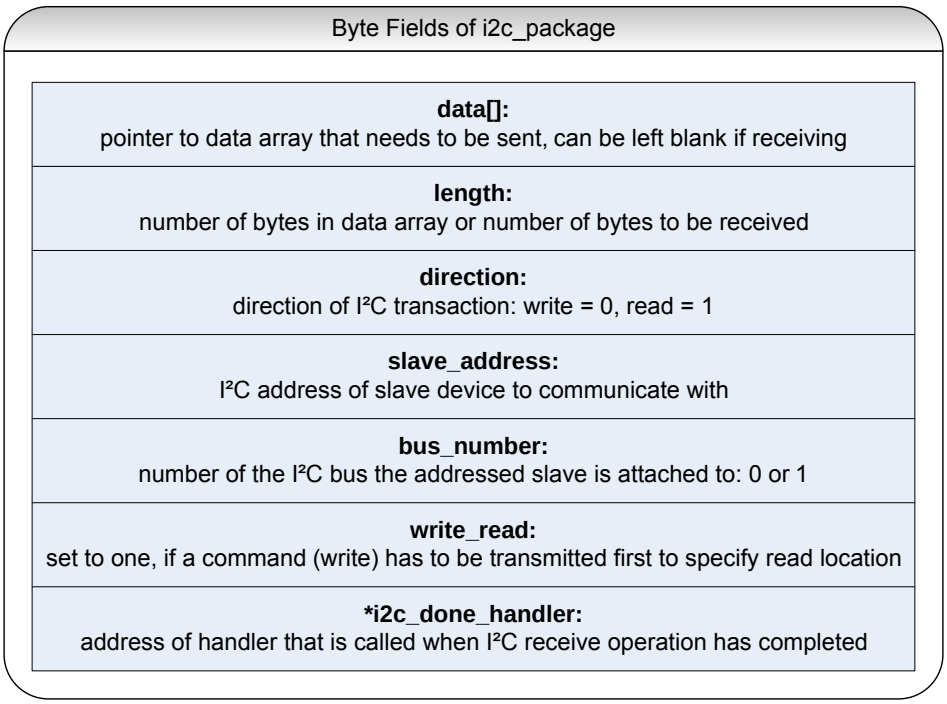


Figure 3.2.: Byte fields of i2c_package

An `i2c_package` has to be passed to the function `i2c_op(i2c_package * package)` to start an I^{2}C transaction. If the direction byte of the `i2c_package` is 0, the I^{2}C subsystem will perform a write operation and if it is 1, it will perform a read operation as soon as the I^{2}C resource is available. When a receive operation is completed, the I^{2}C subsystem calls the function registered in the `*i2c_done_handler` byte of the `i2c_package` to handle the received data.

For reading a specific memory location of an I^{2}C slave device a command specifying this memory location has to be written first from the MCU to the I^{2}C slave device since the I^{2}C is only half duplex. Only then data can be read. To ease the implementation of this process in higher level device drivers, the function `i2c_write_read(i2c_package * package_write, i2c_package * package_read)` can be used. This function first writes the contents of the data array of the `package_write` to the slave device, then generates a "repeated start" condition on the I^{2}C and after that reads the number of bytes specified in the length field of `package_read` from the device.

3.3.2. BMP085 Device Driver

A higher level I^{2}C device driver had to be implemented for the *BMP085 digital absolute barometric pressure sensor [4]* from Bosch Sensortec which is attached to the I^{2}C bus 1. The complete code of this device driver can be found in appendix C.4.

The API of this driver consists of five functions:

- `void bmp085_init()`:
  This function initializes the BMP085 pressure sensor. It tests the sensor for proper operation, reads out the factory calibration parameters and stores them in a `bmp085_t` structure which holds all control, measurement and status data of the sensor. The I^{2}C subsystem has to be initialized beforehand; otherwise this will fail.

- `void bmp085_start_pressure_measurement()`:
  This function has to be called by the application software to start a pressure measurement on the BMP085. The BMP085 needs 4.5ms - 13.5ms (depending on the oversampling setting selected by `BMP085_PRESSURE_CONVERSION_MODE`) until the pressure measurement is done. But this function will return immediately as it only starts the measurement and the rest of the process is handled by the device driver controlled by interrupts. Measurement read-out is done by `bmp085_start_measurement_read()` called by the EOC (end of conversion) interrupt.

- `int32_t bmp085_get_pressure()`:
  This function has to be called by the application software to get the most recent pressure value. It immediately returns the absolute barometric pressure in Pascal.

- `void bmp085_start_temp_measurement()`:
  The BMP085 is not only a pressure sensor, it also comprises an absolute temperature sensor. This function has to be called by the application software to start a temperature measurement on the BMP085. The BMP085 needs 4.5ms until the temperature measurement is done. But this function will return immediately as it only starts the measurement and the rest of the process is handled by the device driver controlled by interrupts. Measurement read-out is done by `bmp085_start_measurement_read()` called by the EOC interrupt. This function needs to be called once before starting the first pressure measurement.

- `int16_t bmp085_get_temperature()`:
  This function has to be called by the application software to get the most recent temperature value. It immediately returns the absolute temperature in 0.1°C. This function also needs to be called once before starting the first pressure measurement.

Any function not listed here is for internal use by the device driver only and should not be called by the application software.

3.3.3. EEPROM Device Driver

When looking at the datasheet of the *93AA86 EEPROM from Microchip [54]* integrated on the prototype of the new IMU in detail for programming the device driver, it was realized that it is actually incompatible with the MCU LPC2148. This EEPROM has been adopted from the previous PIXHAWK electronics and it was assumed to be working because no contrary information had been given. But in fact, the 93AA86 EEPROM had not been tested before. The incompatibility of the EEPROM and the MCU is expressed by the following citations from the datasheets of these devices:

- **Citation from datasheet of 93AA86 EEPROM:**
  The output data bits will toggle on the rising edge of the CLK and are stable after the specified time delay (TPD).
- **Citation from datasheet of LPC2148 MCU:**
  Each bit is driven onto SI line on the falling edge of SK. The SSP in turn latches each bit on the rising edge of SK.

These citations show that the MCU will read data from the MISO line before they are ready because the data only becomes valid TPD after the rising edge. The clock polarity of the SSP interface could be inverted, but then again writing would not work anymore.

For this reason a new EEPROM had to be selected. It was first thought to replace it with a different *SPI EEPROM*. In the evaluation process an issue that all SPI EEPROMs would experience with the interrupt controlled SPI implementation on the LPC2148 was detected. This issue concerns the write process to the EEPROM. The problem is that data does not have to be written to the EEPROMs byte by byte but a continuous write of up to one page can be done. This increases the performance of the EEPROMs but makes it necessary to pull up the chip select line as soon as the last bit has been put on the MOSI line to indicate the end of the write process. Otherwise the EEPROM just continues writing or even discards the data if the chip select line is not pulled up on a byte boundary. But this can not be done with the SSP interface of the LPC2148 in an interrupt controlled manner because it always waits for 32 bus clock cycles after the last byte has been transmitted / received before generating an interrupt such that the device driver can pull up the chip select line. This way the SSP interface can detect the end of an SPI transaction automatically but it also makes it impossible to use it with an EEPROM that supports continuous writes as the chip select line can not be pulled up just at the end of the data transfer.

It would be possible to estimate the end of the SPI transaction and generate a timer interrupt at this time instant which allows a handler function to pull up the chip select line on time. Or the SSP status register could be polled. Anyway, these are just workarounds but no solutions that should be used if other options are available. And the alternative available was to use an I^2C EEPROM instead of an SPI EEPROM. This did not generate too much of additional work as the I^2C subsystem had already been implemented. Therefore I^2C EEPROMs have been evaluated.

The I^2C *EEPROM 24FC256 from Microchip [55]* was selected and added to design for the first IMU hardware revision (see section 5.2). A device driver using the I^2C subsystem has

been implemented. The complete code of the I^{2}C EEPROM driver can be found in appendix C.5.

The API of this driver consists of three functions:

- `void eeprom_write(uint16_t address, uint8_t length, uint8_t * data)`:
  This function writes the `data` byte array to the specified `address` in the EEPROM. `length` specifies the number of bytes in the `data` array (max. 64).
- `void eeprom_start_read(uint16_t address, uint8_t length)`:
  This function starts a read-out of `length` bytes from the specified `address` in the EEPROM. It does not return any data. `eeprom_read_data()` needs to be called after invoking this function to collect the data read out from the EEPROM. Since it will take a few microseconds to read out the data from the EEPROM, other tasks can be executed before collecting the read results, but no other EEPROM requests should be started during this period.
- `void eeprom_read_data(uint8_t * data)`:
  This function collects the data read out from the EEPROM by `eeprom_start_read()` in the byte array `data`. It has to be called after `eeprom_start_read()` and waits for the completion of the EEPROM read by the I2C subsystem. This function returns when all the data read from the EEPROM is in the byte array `data`.

3.3.4. I^{2}C Motor Controller Device Driver

A device driver for the new digital I^{2}C motor controllers has been implemented even though the development of these motor controllers has not yet been completed. This means that the implementation of the I^{2}C slave interface on the motor controller will have to adhere to the communication structure described here. The motor controllers will be attached to I^{2}C bus 0.

The API of the I^{2}C motor controller driver consists of two functions:

- `void motor_i2c_set_rpm(uint8_t mot_i2c_dev_addr, uint16_t rpm)`:
  This function sends the RPM value specified in `rpm` to the motor controller specified by `mot_i2c_dev_addr`. `mot_i2c_dev_addr` is the slave address of one of the I^{2}C motor controllers. Four slave addresses have been defined in the platform configuration `file imu_conf_vxxx.h`[1] as `MOTx_I2C_SLAVE_ADDRESS`[2]. Figure 3.3 shows the complete frame that will be transmitted on the I^{2}C data line (SDA).
- `uint16_t motor_i2c_get_rpm(uint16_t mot_i2c_dev_addr)`:
  This function reads the current RPM value from the motor controller specified by `mot_i2c_dev_addr`. For this purpose the command byte `MOT_GET_RPM_CMD` has been specified to tell the I^{2}C motor controller to reply with the current RPM value after a repeated start[3] condition on the I^{2}C bus. The complete data transaction on the I^{2}C bus triggered by this function is shown in figure 3.4.

[1] xxx designates the hardware version number, currently defined: 200 and 210.
[2] x is a place holder for the number of the specific motor controller; possible values: {1,2,3,4}.
[3] A repeated start happens if no stop condition is generated at the end of an I^{2}C bus transaction, but instead the next transaction is started immediately with a start condition. This indicates to the slave device that these transactions belong together.

The complete code of this device driver can be found in appendix C.6.

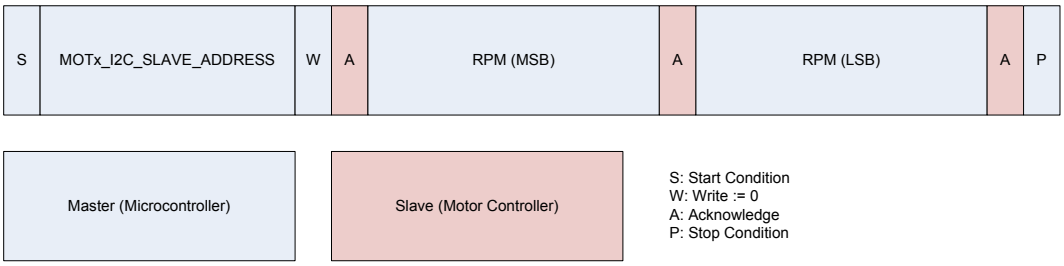


Figure 3.3.: Complete I^{2}C bus transaction to set new RPM value

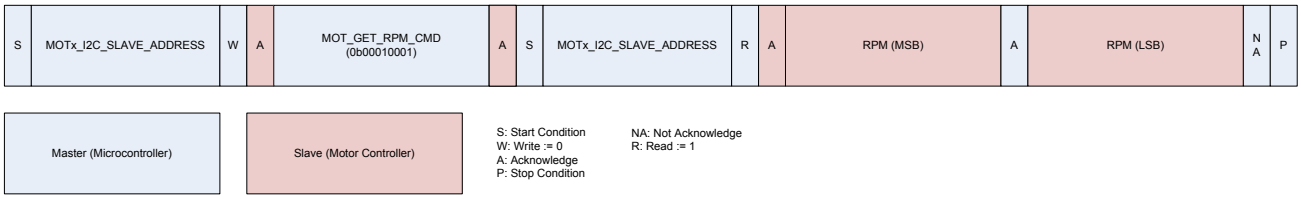


Figure 3.4.: Complete I^{2}C bus transaction to get current RPM value

3.3.5. SCA3100 Device Driver

Since the obsolete SCA3000 accelerometer had to be replaced by the SCA3100 [2], a new driver had to be implemented for this accelerometer. The device driver uses the already implemented SPI subsystem.

The SCA3100 provides great progress over the SCA3000 in terms of driver design possibilities because it automatically transmits status and error bits in the first byte after being selected. This helps to identify and handle error conditions immediately and correctly. The available error handling is completely implemented by this driver and provides the user with specific error messages over the wireless serial link `COMM_1`.

The API of the SCA3100 device driver consists of three functions:

- `void sca3100_init()`:
  This function initializes the SCA3100 linear accelerometer and sets the chip select pin into operational mode. The SPI subsystem has to be initialized beforehand; otherwise this will fail.

- `void sca3100_read_res()`:
  This function starts the read operation of the SCA3100 sensor values over the SPI bus. The SPI subsystem will complete the data transfer and the measurements are temporarily saved in the SCA3100 driver. `while (spi_running()) ;` can be executed to wait for the results. Afterwards `sca3100_get_value(axis)` can be called to retrieve the measurements read out of the sensor.

- `int sca3100_get_value(int axis)`:
  This function retrieves the most recent measurement of axis `axis` from the SCA3100 device driver. The possible values for `axis` are: `SCA3100_X_AXIS`, `SCA3100_Y_AXIS` and `SCA3100_Z_AXIS`. To read out the current values from the sensor, `sca3100_read_res()` has to be executed first.

The complete code of this device driver can be found in appendix C.8. Any function not listed here is for internal use by the device driver only and should not be called by the application software.

4. Tests

This chapter describes the tests with the different prototype boards and shows their results. The first section discusses the functional testing of the different boards. The second and third section show the tests conducted on the IMU and the COM Express SBC electronics respectively to asses the main characteristics of these units.

4.1. Functional Tests

All parts have been electrically tested during the assembly of the boards. The NEC μPD720114 USB hub chip is the only part that failed in the electrical tests for the reasons explained in section 2.3.3. All the other components have been tested for correct operation.

4.1.1. Results of Functional Tests of the Power Supply Prototype

The power supply prototype has been used to power the new IMU and the Gumstix Overo SBC. No issues could be detected. Also the wireless communication between the IMU and the groundstation via the XBee module mounted on the power supply works seamlessly. The design of the servo controller has been used extensively before and proven to be robust. This means, that the power supply prototype has passed the functional tests completely and no parts need to be exchanged in the hardware revision.

4.1.2. Results of Functional Tests of the IMU Prototype

The LPC2148 MCU on the IMU has been programmed with the embedded software described in chapter 3. This software should read out all the sensors at a rate of up to 200Hz and send the measurements via the XBee wireless module to the PIXHAWK groundstation. This works for all the sensors except for the MS2100 magnetosensor from PNI.

The three values for the three measurement axes of the MS2100 permanently stay zero. The cause of this issue could lie in the software or in the hardware. For this reason the chip select, the serial clock and the MOSI pad of the MS2100 footprint have been probed with an oscilloscope. The correct signals could be found on these pins which means that the device driver running on the microcontroller starts the measurement correctly. Then the "data ready (DRDY)" and the MISO pad were tested with the oscilloscope. No signal could be found on these pins which implies that the sensor just does not do anything. The reason for this could either be that the chip is defect or that its pins are not properly soldered to the PCB. Since this issue appears with both IMU prototypes assembled and the maximum allowed solder temperature was not exceeded when soldering the sensors in the reflow oven, it is very unlikely that the chips are defect.

As already mentioned in section 2.2.2.4 the MS2100 has already caused a lot of problems with soldering before. In fact, only one[1] chip could be correctly soldered till now. The reason to select this chip again for the new design was the lack of proper alternatives and the assumption that the use of a reflow oven would eliminate the soldering issues. This experience shows that the assumption was wrong. A lot of time has been invested to develop and try new methods to mount the MS2100 magnetosensor correctly on the PCB. But everything failed. Therefore it was decided to exchange the MS2100 with the 3D magnetometer HMC5843 [27] from Honeywell. This has been done in the first hardware revision as described in section 5.2.

The other sensor measurements have been fed to the existing sensor fusion algorithm which determines the roll, pitch and yaw angles of the IMU's orientation. Since the magnetometer was not working only the roll and the pitch angle can be correctly determined. Figure 4.1 depicts the roll and pitch output at zero degree inclination. Figure 4.2 shows the roll output at different inclinations about the X axis and figure 4.3 shows the pitch angle at different inclinations about the Y axis. This can only be rated as a functional test because the inclination angles have been set manually. This means that the output deviations from the nominal angle in the determined magnitude can also be caused by the manual positioning. Our project team does not own tools for more precise angular positioning. But the accuracy shown in figures 4.1 to 4.3 is already good enough for controlling the MAV and this test can be used as a proof of concept. If the positioning was more precise, the deviations would be even smaller.

As described in section 3.3.3 the EEPROM 93AA86 from Microchip integrated on the IMU prototype is incompatible with the microcontroller LPC2148. Therefore it is replaced by the EEPROM 24FC256 from Microchip in the first hardware revision presented in section 5.2.

4.1.3. Results of Functional Tests of the COM Express Baseboard Prototype

The prototype of the COM Express Baseboard has been extensively used with a wide variety of peripherals to test the interfaces for proper operation. The results are shown in this section.

- **Fully operational:**
  - 18.5V and 11.1V input for 12V power rail
  - 5V power supply
  - 3.7V power supply
  - COM Module
  - all USB Ports
  - all UART Ports
  - I^2C Port
  - all GPIOs
  - VGA Port
  - Servo Controller

[1] 1 out of 15

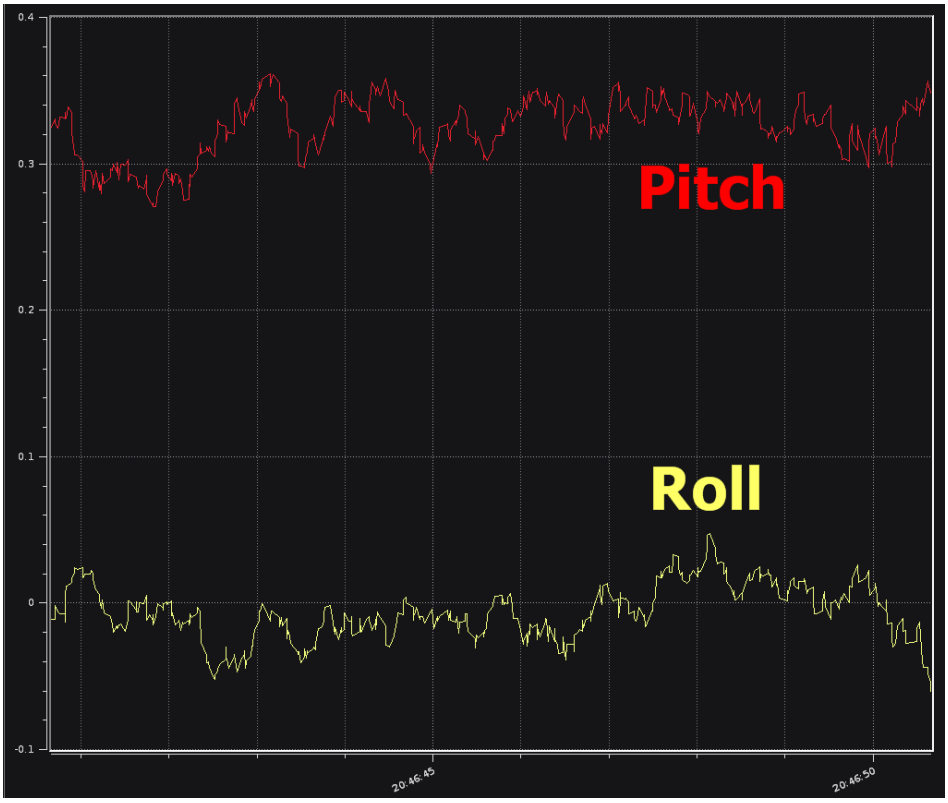
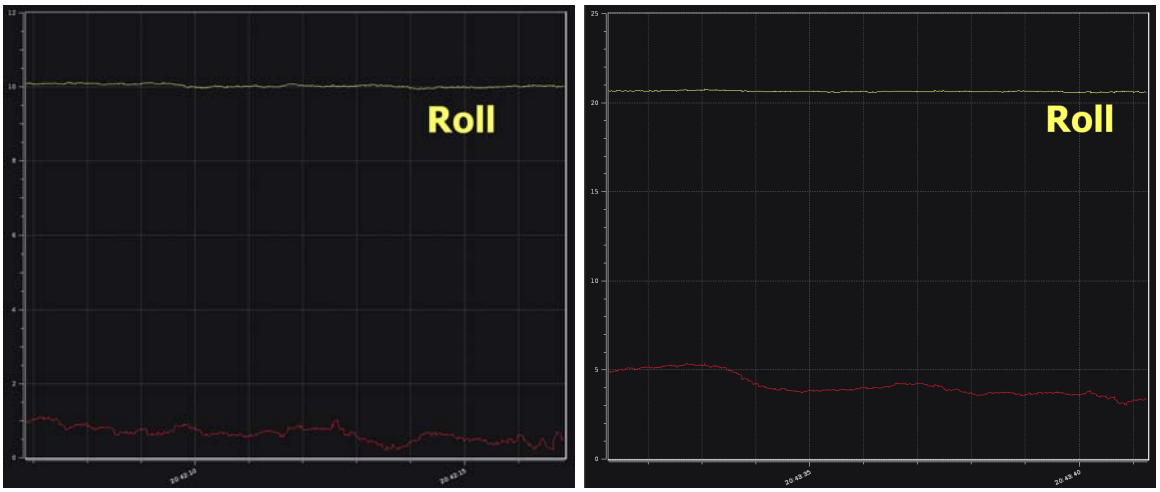
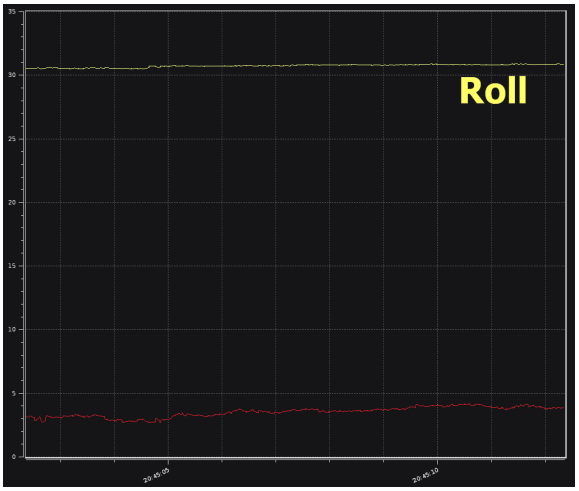


Figure 4.1.: Roll and pitch output of the sensor fusion at 0° inclination

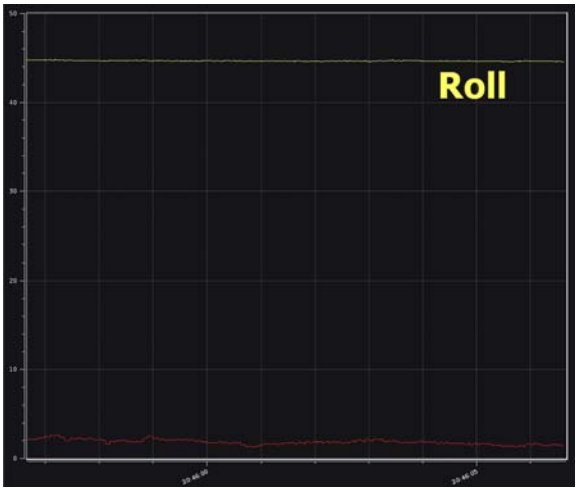


(a) 10° inclination

(b) 20° inclination

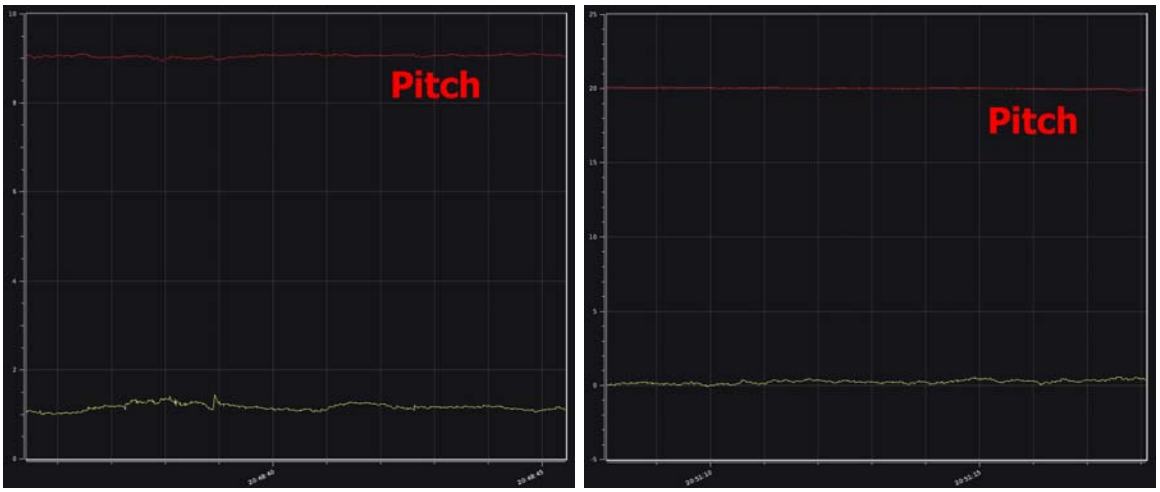


(c) 30° inclination



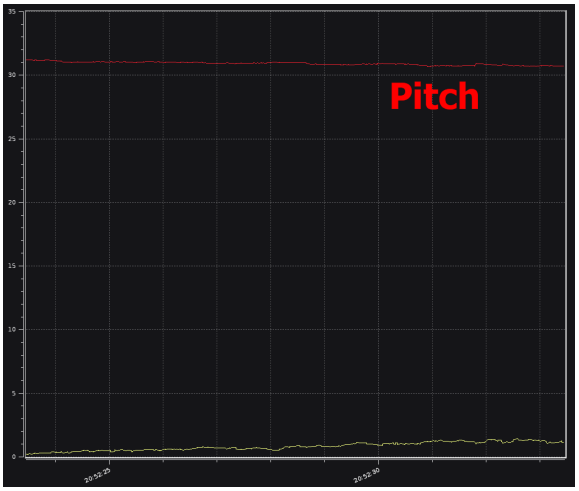
(d) 45° inclination

Figure 4.2.: Roll output of the sensor fusion at different inclinations about the X axis

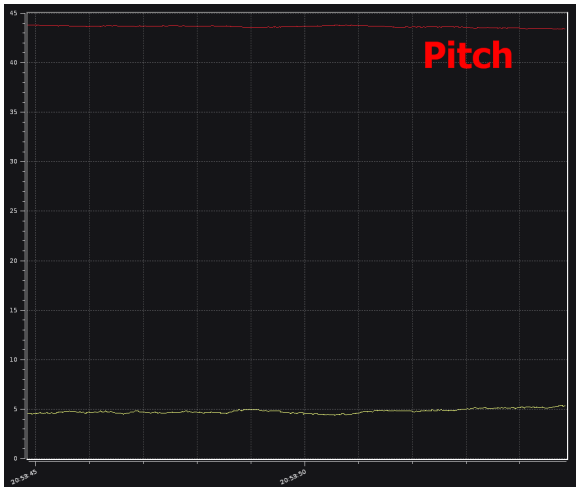


(a) 10° inclination

(b) 20° inclination



(c) 30° inclination



(d) 45° inclination

Figure 4.3.: Pitch output of the sensor fusion at different inclinations about the Y axis

- **Partly operational**
  - 3.3V power rail:
    It is under-powered. One LDO cannot provide enough power, if all peripherals are connected. But it can provide enough power for all permanent on-board components.
  - XBee module:
    It turns off during transmissions, because of the under-powered 3.3V rail. If externally powered, it works fine.
  - SATA II Interface:
    The SATA II data interface works perfectly, but the power connector cannot source enough current on the 3.3V rail to drive an SATA SSD because of the underpowered 3.3V rail. If the SATA II peripheral is powered externally, it works as expected.
- **Not operational:**
  - HDMI:
    The HDMI interface is not working because no HDMI level shifter has been integrated on the COM Express baseboard prototype, which would be necessary for proper operation. The preliminary datasheet of the COM module manufacturer available at the design time did not specify whether the HDMI level shifter was already integrated on the COM module or whether it had to be implemented on the baseboard. Therefore an inquiry regarding this matter was sent to the manufacturer. Unfortunately the reply from the manufacturer contained the faulty information that the HDMI signals can be routed directly to the connector without an HDMI level shifter in between. This information has later been revised by the manufacturer upon a second inquiry.

The two issues exhibited by the functional tests are resolved in the first hardware revision of the COM Express baseboard described in section 5.4.

The COM Express SBC has also be mounted and tested in flight by John Stowers on a commercial, customizable quadrocopter. The SBC did not show any failures during these tests, i.e. its power supply supplies a stable input voltage even if the motors draw large and strongly varying currents from the same battery.

4.1.4. Results of Functional Tests of the USB-to-UART Adapter

The USB-to-UART adapter has passed all functional tests and does not need to be revised.

4.2. IMU Testing

This section describes a series of tests conducted on the PIXHAWK IMU prototype to asses the most important characteristics of the sensor array.

4.2.1. Static Sensor Offset and Noise Distribution Identification

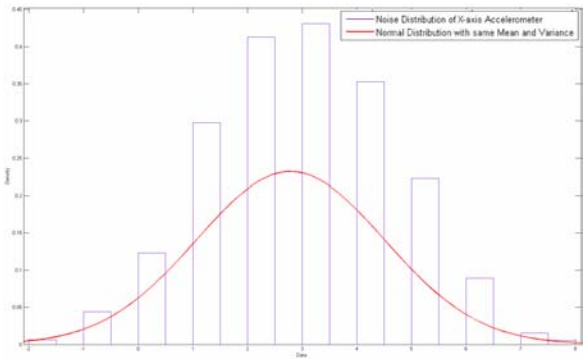
This test was conducted to determine the static sensor values at 0° inclination about all three axis and 0°/s angular velocity. This is necessary to set a *reference orientation* and the respective sensor values in software. For this purpose the IMU was laid flat on a table and the sensors values were logged over about 30 seconds.

Naturally the recorded sensor values are not static but disturbed by measurement noise. The distribution of the measured sensor values has been plotted with Matlab for each sensor and axis. Figure 4.4 shows the noise distribution for each sensor value compared to a normal distribution with the same mean and variance. The mean of each recorded sensor value can be used to set the static reference orientation at 0° inclination about all three axis and 0°/s angular velocity and is shown in table 4.1. The variance and the maximum measured deviation assess the accuracy of each sensor value. This is also shown in table 4.1.

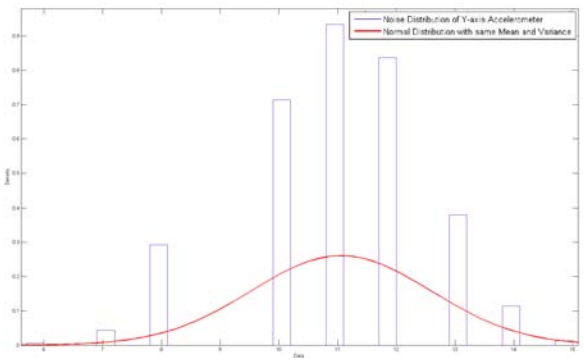
Sensor	**Mean**	**Variance**	**Mean Accuracy**	**Min. Accuracy**
X-Axis Acc.	2.78	2.955	±1.478 mg	5 mg
Y-Axis Acc.	11.06	2.337	±1.169 mg	5 mg
Z-Axis Acc.	1021.4	3.371	±1.686 mg	7 mg
Phi Gyro	29683	17.53	±0.134 °/s	±0.244 °/s
Theta Gyro	30232	19.1	±0.146 °/s	±0.198 °/s
Psi Gyro	29809	14.24	±0.109 °/s	±0.198 °/s
Pressure	97397 Pa	42.69 Pa	±21.35 Pa	±22 Pa

Table 4.1.: Static sensor offsets in reference orientation and sensor accuracy

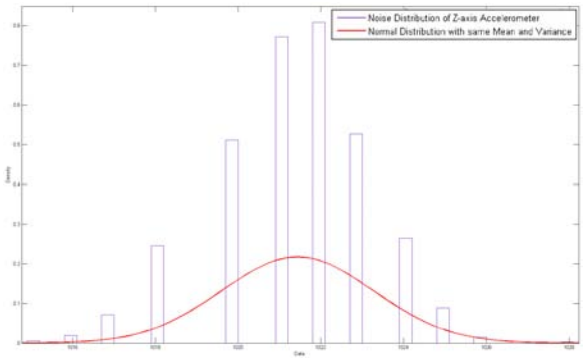
The accelerometer and the gyros exhibit a good accuracy. The accuracy of the pressure sensor is not good since ±22Pa corresponds to about ±1.75m at sea level. It has to be said that the BMP085 pressure sensor was used in high-speed, low-accuracy mode when conducting this test to receive the raw data from the sensor. To improve the accuracy either the sensor can be set to operate in low-speed, high-accuracy mode in which it performs internal oversampling itself or the oversampling can be performed on the MCU with a sliding window to improve the accuracy while still maintaining a continuous pressure output.



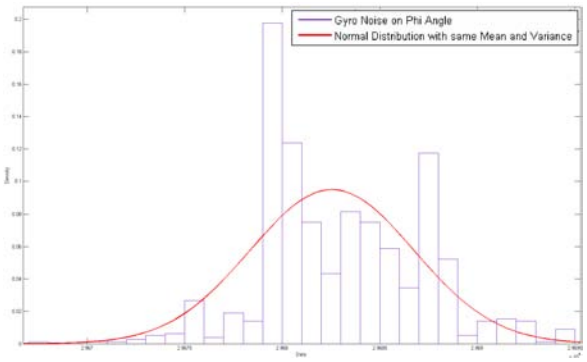
(a) X-Axis Accelerometer



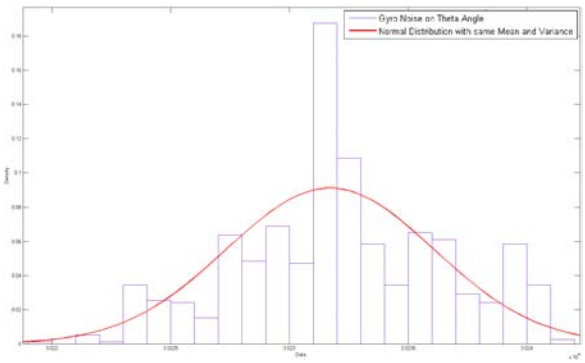
(b) Y-Axis Accelerometer



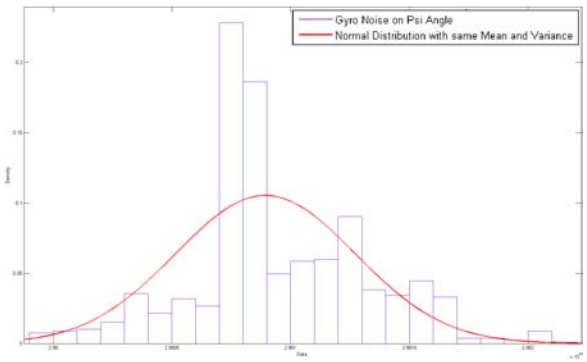
(c) Z-Axis Accelerometer



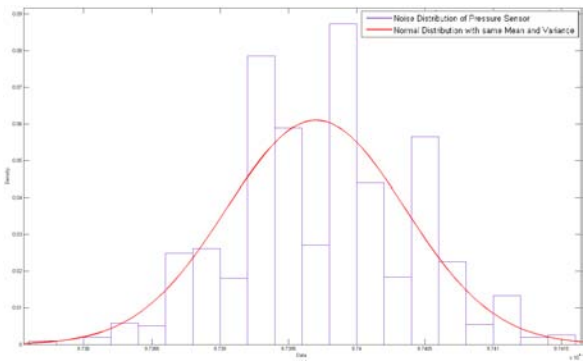
(d) Phi Gyro



(e) Theta Gyro



(f) Psi Gyro



(g) Pressure Sensor

Figure 4.4.: Sensor measurement noise distribution in static reference orientation

4.2.2. Scale Factor Determination for Gyroscopic Angular Velocity Sensors

Due to the conditional differences of manufacturing of the MEMS gyroscopes, the angular velocity scale varies a bit from chip to chip. Therefore the scale factor determining the relation between the gyro sensor value and the current angular velocity has to be determined for all three orientation angles. Equation 4.1 shows the relationship between angular velocity ω (in rad/s), the gyro sensor value g, the scale factor s and the static sensor value o:

$$\omega = (g - o) * s \tag{4.1}$$

To determine the scale factor s, the IMU was positioned on the rotating plane of a record player. The record player has an angular rate of 45 rotations per minute which corresponds to 4.7124rad/s. The gyro values at this angular velocity have been logged for about 30 seconds. Table 4.2 shows the mean of the three gyro values at 45 rotations per minute. From these measurements, the scale factor s can be derived by using equation 4.1 and the static sensor values shown in table 4.1. Table 4.2 lists the scale factors for the three sensor axes.

Axis	**Mean at 45 rpm**	**Scale Factor s**
Phi Gyro	34617	0.000955 rad/s/Gyro Step
Theta Gyro	35958	0.000823 rad/s/Gyro Step
Psi Gyro	34475	0.001010 rad/s/Gyro Step

Table 4.2.: Gyro sensor values at 45rpm and derived gyro scale factors

4.2.3. Temperature Dependency of Gyro Sensors

The SCA-3100 3D accelerometer from VTI does temperature compensation of the acceleration sensing elements internally. For the BMP085 pressure sensor the temperature compensation was provided by the manufacturer Bosch Sensortec in form of C code and is implemented in the device driver. But for the gyroscopes IDG-500 and ISZ-500 from Invensense no temperature drift model was available. Temperature drift compensation is particularly important for the gyros because these sensors usually show the strongest temperature drift of all sensors of an IMU. Therefore the following test was conducted to asses a temperature drift model for all three rotation angles.

For this purpose the IMU was put in an oven and heated up to 50°C board temperature. Then logging of the gyro values was started and the IMU was let to cool down to 35°C indoors. At 35°C logging was stopped and the IMU was put outside. Logging was restarted and the gyro values were logged for 30°C down to 10°C. After this the IMU was heated up again to 42°C and then placed in front of an open window. Logging of the gyro sensors was then done between 40°C and 20°C. At the end all logs were combined.

The logging process was done in three steps for high, medium and low temperatures because the temperature gradient on the board at the start of each measurement is not as large as if the board was put outside directly at a temperature of 50°C. This means that the log contains many more gyro values also for high temperatures which increases the reliability of the measured data.

The log was imported into Matlab. The logged sensor values of each of the three gyros was plotted against the the temperature log as shown in figures 4.5 to 4.7. Then two curves were fitted for each of the three datasets with polynomial curve fitting. The result of the curve fitting can also be seen in figures 4.5 to 4.7.

The first curve (red) is a linear polynomial fit to approximate the temperature drift linearly between 25°C and 40°C. This curve can be used to easily implement temperature drift compensation in software. It is expected that the board temperature will stay within the given temperature boundaries in most use cases as the operating temperature of the IMU lies between 30°C and 35°C indoors. It can clearly be seen that the linear curve approximates the temperature drift very well in the specified temperature range for all sensors. For the Theta gyro the linear approximation is also exact over the complete measured temperature range.

The second curve (blue) is a quadratic polynomial fit to approximate the temperature drift of the gyros over the full recorded temperature range. Figures 4.5 to 4.7 show that this quadratic polynomial follows the sensor values closely over the complete measured temperature range.

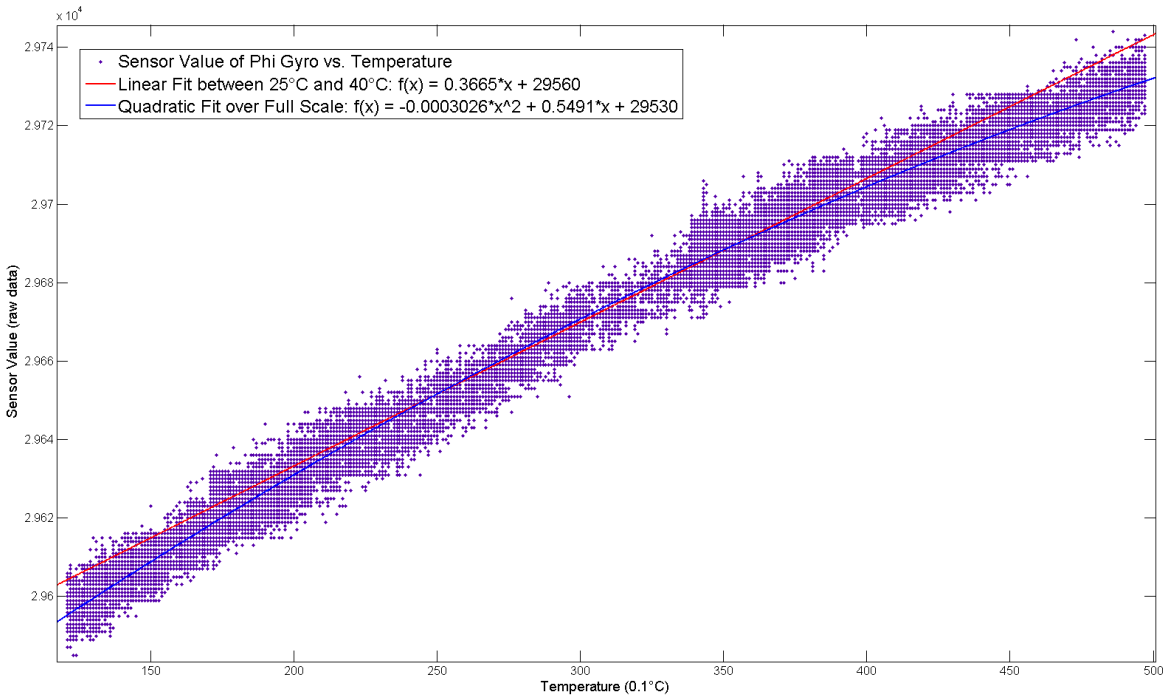


Figure 4.5.: Temperature dependency of Phi Gyro

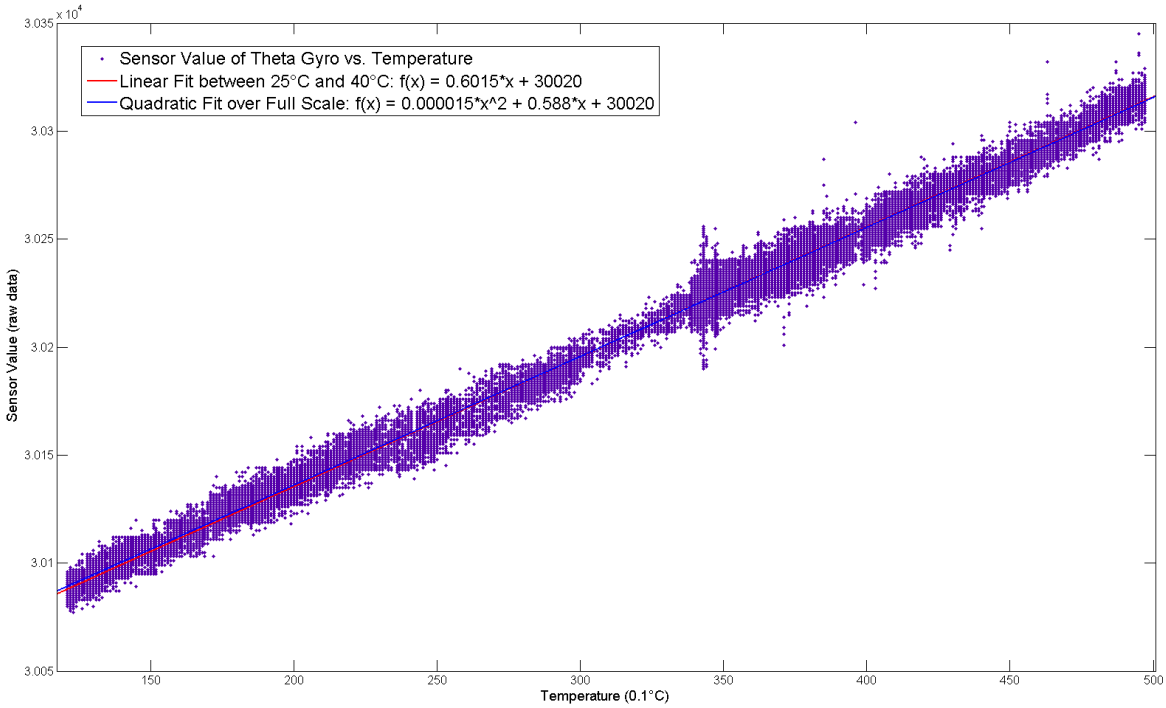


Figure 4.6.: Temperature dependency of Theta Gyro

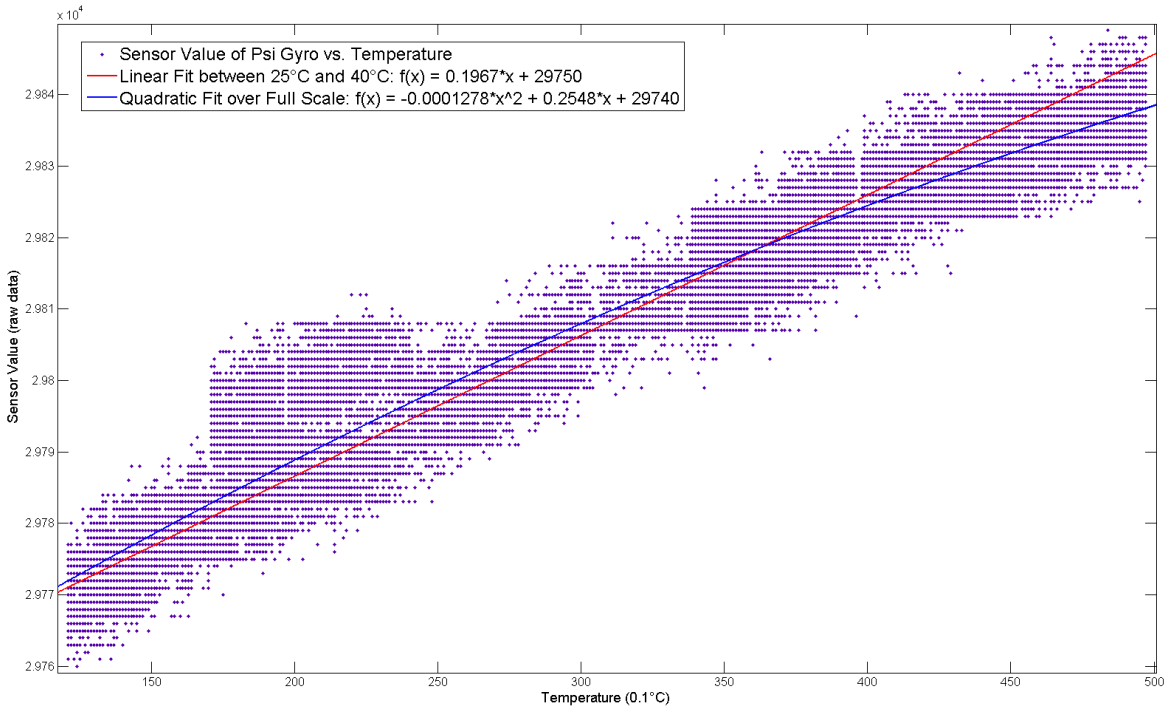


Figure 4.7.: Temperature dependency of Psi Gyro

4.3. COM Express SBC Electronics Testing

This section describes a series of tests conducted on the PIXHAWK COM Express SBC electronics with the *Hardware Analysis and Benchmarking* tool SiSoftware Sandra 2010 to asses the performance of the new on-board computer.

4.3.1. CPU Benchmark

The SiSoftware Sandra 2010 arithmetics (ALU, FPU, etc.) and multimedia (SSE, MMX, etc.) benchmarks were conducted with the Intel Core 2 Duo SL9400 CPU integrated on the selected COM Express module. The benchmark results are shown in figures 4.8 and 4.9. The score of the Intel Core 2 Duo SL9400 CPU is represented by the red dot. It is compared to 4 other CPUs whose comparison is of interest. The performance profiles of these CPUs are integrated in the SiSoftware Sandra benchmark tool.

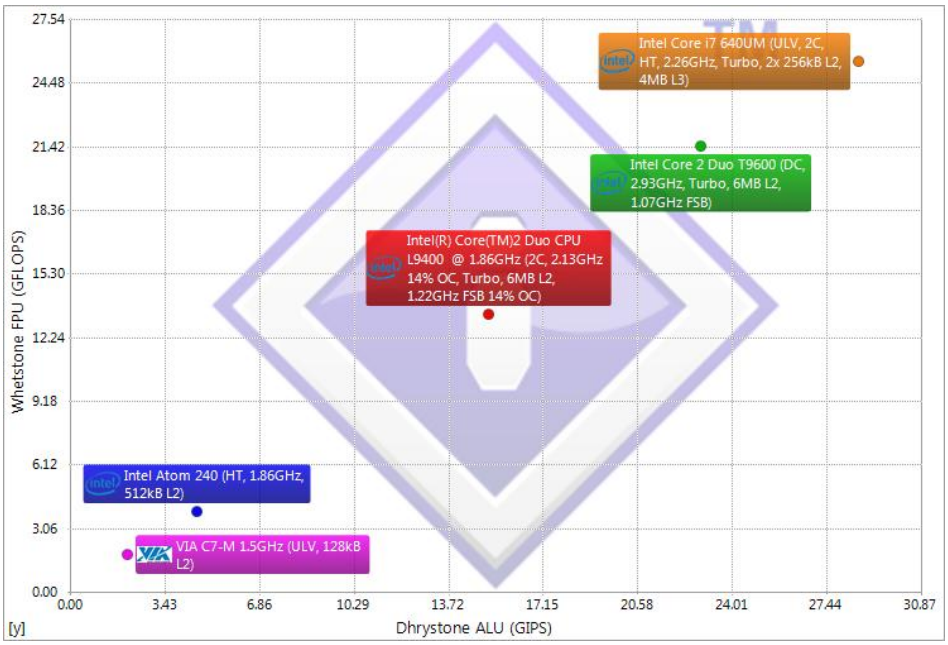


Figure 4.8.: Result of arithmetic performance benchmark conducted on the SBC electronics

The VIA C7-M (pink dot) was a CPU option when selecting the standard COM form factor of the new SBC. It is by far the slowest CPU in all respects which confirms the deselection of the platforms featuring the VIA C7-M CPU.

The Intel Atom 240 (blue dot) was a CPU option for even smaller COM form factors than the selected COM Express platform. This is the most interesting comparison since other competing research teams in the field of MAVs are developing on-board computers based on Atom processors. The Atom 240 runs at the same clock frequency of 1.86GHz as the Core 2 Duo SL9400, therefore the comparison should be fair. Table 4.3 lists the speed-up (performance multiplication factor) in the four different aspects tested by the benchmarks.

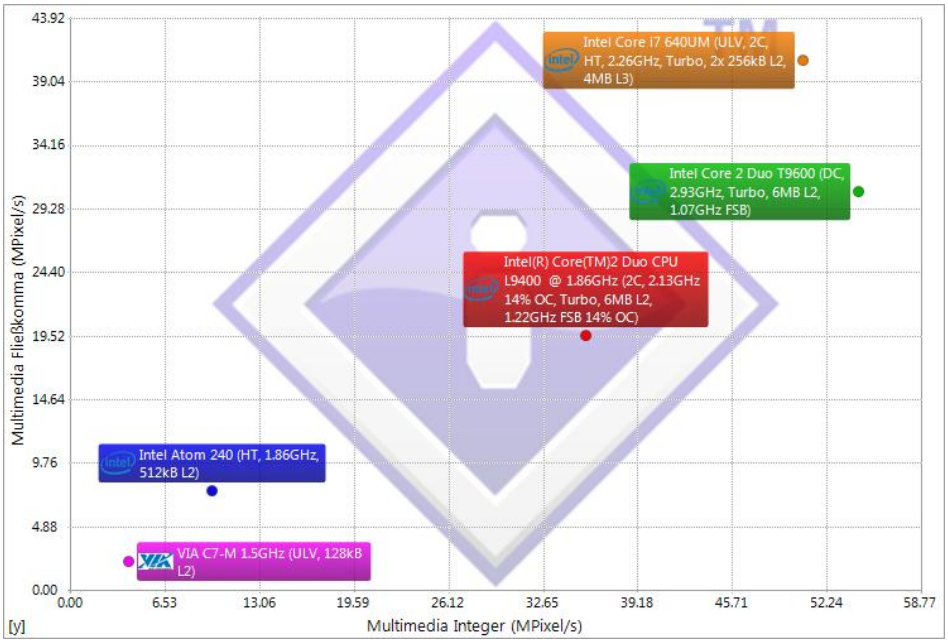


Figure 4.9.: Result of multimedia performance benchmark conducted on the SBC electronics

Benchmark	Speed-up
Arithmetic Integer Performance	3.6
Arithmetic Float Point Performance	3.3
Multimedia Integer Performance	3.7
Multimedia Floating Point Performance	2.6

Table 4.3.: Performance increase factor (speed-up) of Core 2 Duo SL9400 vs. Atom 240

The speed-ups listed in table 4.3 result from synthetic benchmarks using the CPU only. One has to bear in mind that the memory interface of the chipset used on Atom based COMs and also the memory itself is again slower than the respective parts used with the Core 2 Duo. Tests conducted with the PIXHAWK computer vision software on an Intel Core 2 Duo based desktop PC (same CPU architecture, 2.0GHz core clock) have been executed up to 8x faster than on an Intel Atom based system [56].

The Intel Core 2 Duo T9600 (green dot) is one of the top CPUs for notebooks currently available and is based on the same architecture as the Core 2 Duo SL9400. The scores show that top level notebook CPUs outperform low voltage CPUs of the same architecture clearly. But the comparison with the Intel Core i7 640UM (orange dot) which is a CPU based on Intel's new Westmere architecture and has the same power consumption as the Core 2 Duo SL9400 shows that next-generation low voltage CPUs will perform comparably to today's top level notebook CPUs. Because the new SBC electronics are based on a standardized platform, we will profit from this progress without any hardware adaptions being necessary.

Since the comparison with the Intel Atom is the most important one, the same benchmarks have also been conducted on an Intel Atom based computer to confirm the performance data provided by SiSoftware Sandra. The tested computer has an Atom Z540 integrated that runs at 1.86GHz just like the Atom 240. Figures 4.10 and 4.11 depict the results of these benchmarks. The score of the Atom Z540 is nearly identical with the performance profile of the Atom 240 which proves the data provided by the test tool to be reliable.

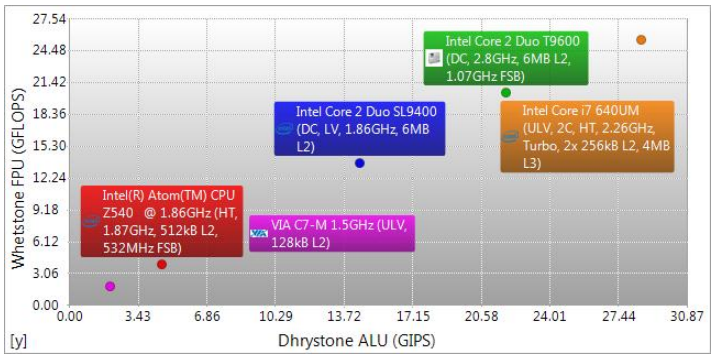


Figure 4.10.: Result of arithmetic performance benchmark conducted on Atom based computer

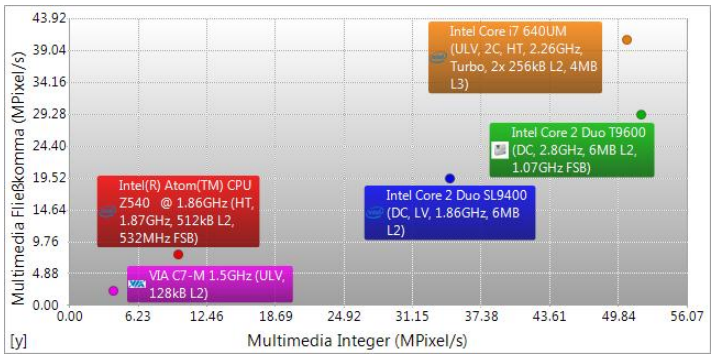


Figure 4.11.: Result of multimedia performance benchmark conducted on Atom based computer

4.3.2. Maximum Power Consumption Measurement

To determine the maximum power consumed by the SBC electronics (COM module + baseboard), the CPU and the GPU burn-in test from SiSoftware Sandra were run in parallel. This puts both CPU cores and the GPU under full load. During this test the current consumption was measured with a Fluke multimeter. Table 4.4 shows the average and the maximum power consumption of the PIXHAWK on-board computer under full load.

Maximum Power Consumption	Average Power Consumption
29.4W (2.A @ 11.3V)	27.1W (2.4A @ 11.3V)

Table 4.4.: Power consumption of the PIXHAWK on-board computer under full load

4.3.3. CPU Temperature Assessment

The COM Express SBC electronics currently has to be used with the heatsink supplied by Kontron for sufficient heat dissipation. This configuration is shown in figure 4.12. With just the heat sink and no air convection the CPU temperature still increases steadily until the CPU turns itself off to protect itself from thermal destruction.

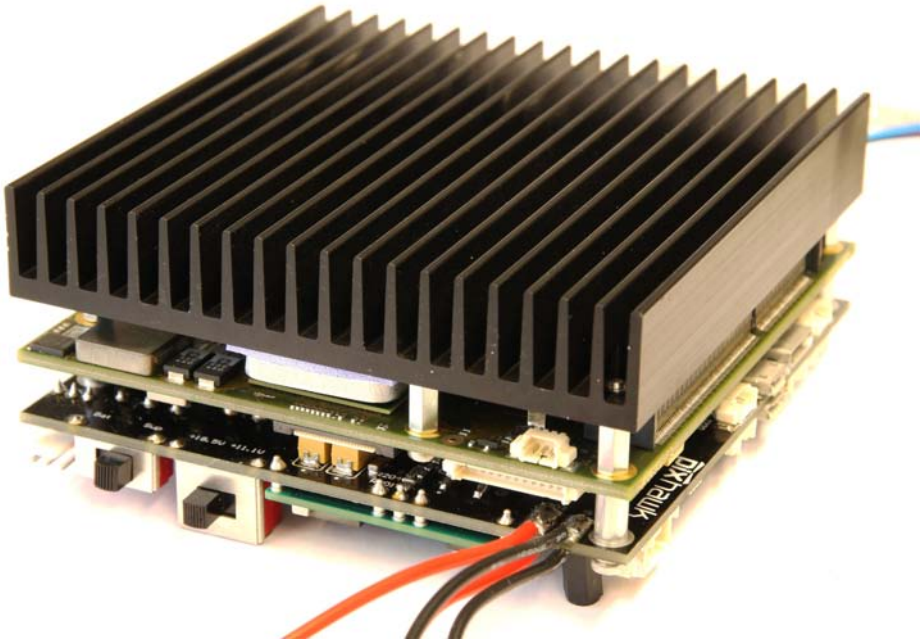


Figure 4.12.: COM Express SBC electronics with Kontron heatsink

The fan supplied with the heatsink is pretty large (60mm) and consumes 360mA @ 12V (4.3W). This power consumption is immense compared to the overall power consumption of the on-board computer under full load of 27.1W. Therefore three smaller and more efficient fans were ordered. To test their cooling effect, the CPU burn-in test from SiSoftware Sandra was run for 15 minutes with one of the new fans mounted.

The test was started with the smallest and lightest fan, the Noiseblocker BlackSilent XM2 shown in figure 4.13. The temperature evolution of the CPU (blue), chipset (green) and GPU (red) temperatures is given by figure 4.14. It shows that the Noiseblocker BlackSilent XM2 can keep the CPU as cool as necessary. Also this fan only consumed 50mA @ 12V (0.6W). Therefore this fan has been immediately selected to be used to cool the COM.

Figure 4.13.: Noiseblocker BlackSilent XM2 40x40x10mm fan [6]

Since the heatsink from Kontron is far to heavy (223g), a new cooling solution using the NB32K chipset cooler (37g) from Zalman is currently being developed. A first prototype of the new cooling solution is depicted in figure 4.15 which also shows the overall weight of the new PIXHAWK on-board computer. The overall weight is decreased from 410g to 235g by the new cooling method. The CPU temperature assessment will have to be redone when this new cooling system is complete.

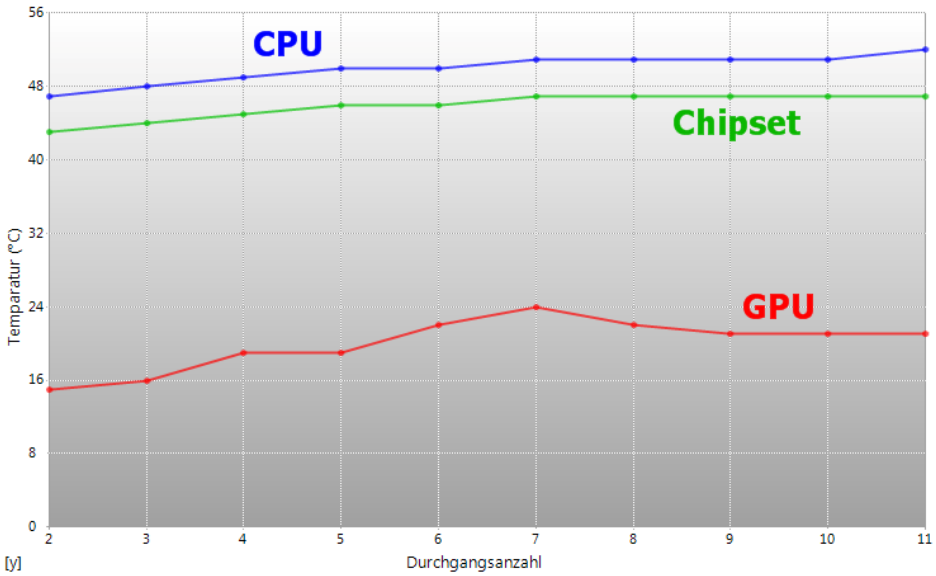


Figure 4.14.: System temperatures during CPU burn-in test

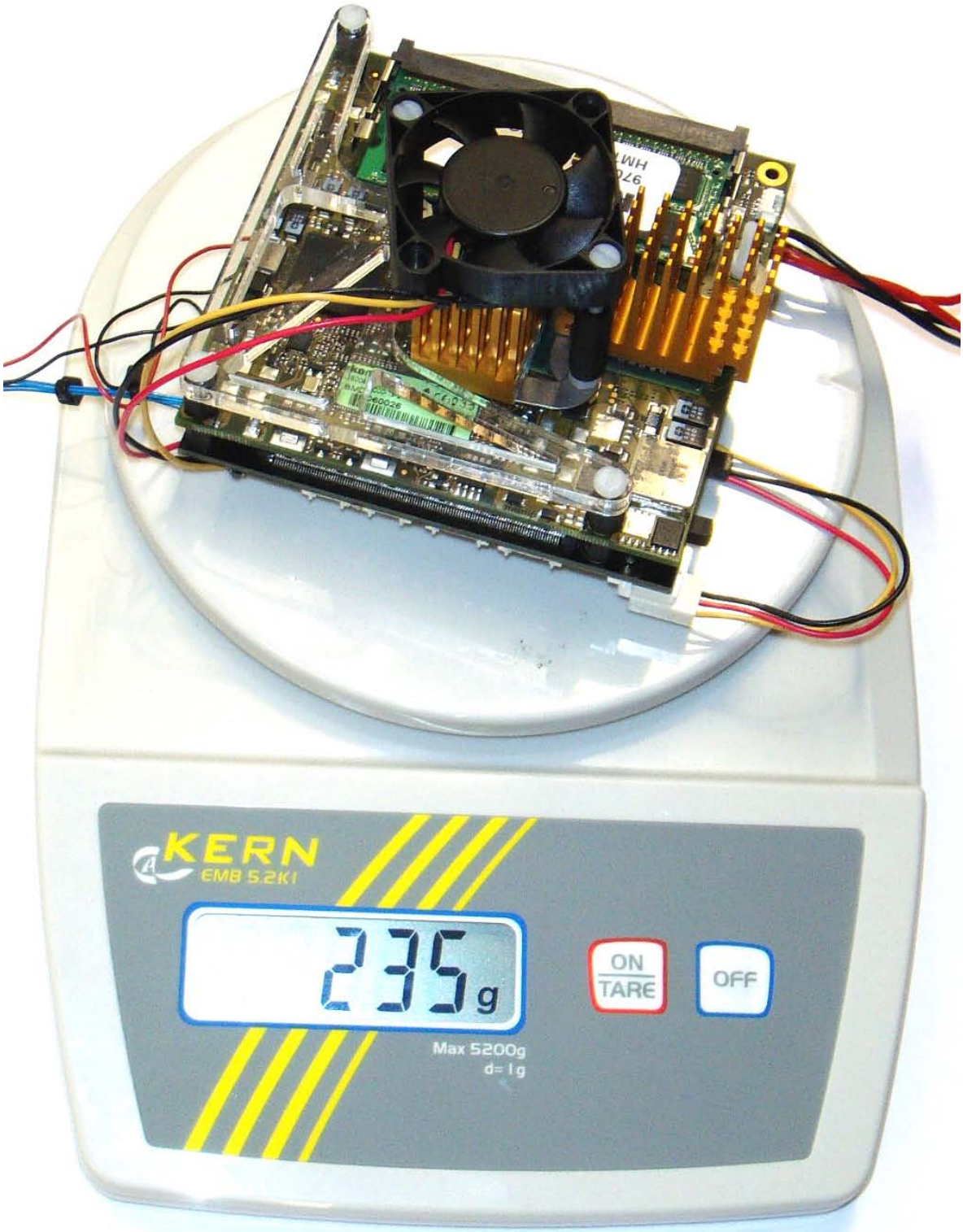


Figure 4.15.: Completely assembled COM Express SBC electronics on the scales showing the overall weight with the new cooling solution

5. Hardware Revision

This chapter describes the changes made to the prototype hardware design shown in chapter 2 during hardware revision. This hardware revision should resolve all functional issues[1] experienced with the prototype hardware and extend the functionality of the prototype hardware where necessary. Test results can not be given yet because the hardware just went into production.

5.1. Power Supply

5.1.1. Hardware Changes

No functional changes had to be made to the power supply during hardware revision. Only one capacitor was repositioned on the PCB layout because of mounting difficulties during assembly of the prototype. Additionally some changes have been made to the silkscreen layers to improve the assembly print on the PCB.

5.1.2. Final Specifications

The final schematics of the power supply can be found in appendix A.1 and a connector description is given in appendix B.1. The final specifications are shown below:

- **Power Input:** 6.0V - 14V DC
- **Power Output:**
  - max. 6A @ 5.0V
  - max. 6A @ 3.7V
- **XBee Pro:**
  - 2.4 GHz wireless module
  - based on ZigBee
  - max. data rate: 115200 bits/s (57600 bits/s bidirectional)
  - max. current consumption: 295 mA @ 3.3V (transmit current)
  - range: 100 m indoors, 1.6 km outdoors

[1] except for the USB hub on the Overo baseboard

- **Servo Controller:**
  - 5 Stage Johnson Decade Counter NXP HEF4017B for PWM control of servos
  - connectors for up to 6 servos
- **Motor Controllers:** connectors for 2 digital I^2C motor controllers
- **Size of PCB:** 40 x 32 mm
- **Weight (completely assembled):** 18g (with XBee module: 22g)

5.2. Inertial Measurement Unit

5.2.1. Hardware Changes

The major hardware change on the revised IMU is the replacement of the Microwire EEPROM 93AA86 [54] with the I^2C EEPROM 24FC256 [55] from Microchip because of the incompatibility of the 93AA86 with the microcontroller LPC2148 described in section 3.3.3. Other minor changes include the addition of a third LED and the adaptation of the LED color assignment to the Twog board of the Paparazzi project [14]. Additionally the EOC (end of conversion) signal of the BMP085 pressure sensor was moved to pin 31 of the MCU which can be configured as the source for the external interrupt 2 (EINT2). This allows for perfect synchronization of the sensor with the embedded software running on the MCU and has already been implemented in the device driver of the BMP085.

Finally the 3D magnetometer MS2100 from PNI had to be replaced because no way was found to solder this chip correctly. It had been a hard decision to replace this sensor only because of the bad design of its package which just makes it nearly impossible to manually solder it. But it has not been working and a non-functional chip has to be replaced for whatever reason. Therefore it was replaced with the 3D magnetometer HMC5843 from Honeywell, which had been rated as the second best compass IC available during sensor evaluation (see section 2.2.2.4). This is in fact the second major change of the revised IMU hardware.

5.2.2. Summary of Changes

- **EEPROM:** Microwire EEPROM Microchip 93AA86 replaced by I^2C EEPROM Microchip 24FC256
- **3D magnetometer:** PNI MS2100 replaced by Honeywell HMC5843
- **BMP085 (pressure sensor):** EOC signal moved to external interrupt 2 (EINT2)
- **LEDs:** third LED added, color assignment adapted to Twog board
- **Boot pin:** boot pad added (pull to GND to boot into ISP mode)

5.2.3. Final Specifications

The final schematics of the IMU can be found in appendix A.2 and a connector description is given in appendix B.2. The final specifications are shown below:

- **Power Input:** 3.7V DC
- **Power Consumption:** max. 120mA @ 3.7V ≈ 450mW
- **Microcontroller:**
  - Part Number: NXP LPC2148 [15]
  - 16-bit/32-bit ARM7TDMI-S microcontroller
  - Max. Clock Frequency: 60 MHz
- **Sensing Capabilities:**
  - 3D Linear Acceleration (VTI SCA3100-D04 [2]):
    - Resolution: 1.11 mg
    - Range: ±2g
  - 3D Angular Velocity (Invensense IDG-500 [3] and ISZ-500 [24]):
    - Resolution: 0.1 deg/s
    - Range: ±500 deg/s
  - 3D Earth magnetic field (Honeywell HMC5843 [27]):
    - Resolution: 0.034 μT
    - Range: ±450 μT
  - 1D Barometric Pressure (Bosch BMP085 [4]):
    - Resolution: 0.08 m
    - Range: -500 m at 110 kPa (101.325 kPa at mean sea level) to about +9000 m at 30 kPa
  - 1D Temperature (Bosch BMP085 [4]):
    - Resolution: 0.1 °C
    - Range: -20 - 65 °C
- **External Interfaces:**
  - 1x USB Micro-B
  - 1x SPI with 3 chip select lines
  - 1x I^2C plus one GPIO pin
  - 4x GPIO pin configurable as ADC input
  - 1x UART interface to SBC
  - 1x interface to power supply (contains signals for servo / motor controllers, UART interface to XBee, battery sensing and power lines)
- **Size of PCB:** 40 x 30.2 mm
- **Weight (completely assembled):** 8g

5.3. Baseboard for Gumstix Overo SBC

5.3.1. Hardware Changes

Since the Overo SBC electronics have been superseded by the COM Express SBC electronics, the hardware revision of this PCB has been done at the very end of the master thesis. It was looked for an alternative USB hub chip to replace the μPD720114 [37] from NEC, that has suffered from permanent failure (see section 2.3.3). Both already evaluated USB hub chips are no good options because of the following reasons:

- **SMSC USB2514 [38]:** Regarding the feature set, the size and the power consumption the USB2514 could replace the μPD720114 one-to-one. The problem of this chip is its package. It only has one ground pin that is located in the center on the bottom of the chip. This pin is not accessible when the chip is mounted, i.e it is impossible to solder this chip manually. This chip can only be used in series production.

- **ST Ericsson ISP1520 [39]:** This chip is much larger than the μPD720114. The Overo baseboard PCB had not been designed to accommodate a hub chip of this size. A major redesign of the layout, probably also an increase of the PCB size, would be necessary when using this chip. Additionally the typical current consumption of 288mA of the ISP1520 if all four ports are used is critical with the current 3.3V power supply.

Because of these reasons, it has been looked for other USB hub chips. Only one more USB 2.0 hub IC available to end costumers could be found, namely the CY7C65640A [57] from Cypress Semiconductor. This hub chip would be a good replacement regarding the size and feature set. The package is also acceptable. But the typical current consumption of 460mA if all four ports are used is just way too much for the small 3.3V LDO used on the Overo baseboard. A physically notably larger LDO would need to be used to power this hub chip. This would again require an extensive redesign of the Overo baseboard PCB.

Due to the effort necessary for implementing a replacement solution for the μPD720114 USB hub controller, this task could not be completed within the time frame of this master thesis. For this reason, the μPD720114 and the associated components have been removed from the PCB, the USB-to-UART converter has been attached directly to the High-Speed USB interface of the Overo SBC and the following two minor changes have been applied to make the baseboard fully functional:

1. **ST2329A bidirectional voltage level translator:**
   Pull-up resistors have been added on the TXD lines of UART3. This is necessary for the ST2329A level shifter to operate correctly.

2. **Overo Reset Circuit:**
   The implementation of the reset circuit for the Gumstix Overo SBC had been using the ADG3308 level shifter. This device has failed to sink enough current to reset the SBC. Therefore the reset circuit is now implemented by an enhancement mode PMOS transistor.

5.3.2. Final Specifications

The schematics of the revised baseboard for the Gumstix Overo SBC can be found in appendix A.3 and a connector description is given in appendix B.3. The final specifications are shown below:

- **Power Input:** 3.7V DC and 5.0V DC
- **Power Consumption:** max. 400mA @ 3.7V $\approx$ 1.5W
- **Single Board Computer:** Gumstix Overo Fire
- **External Interfaces:**
  - 1x USB Micro-AB and USB on PicoBlade with USB On-the-Go functionality
  - 1x SPI with 2 chip select lines (shared connector with I^2C)
  - 1x I^2C (shared connector with I^2C)
  - 4x UART interface (available)
  - 1x UART interface for communication with IMU
  - 1x UART interface connected with console
  - 2x LED on GPIO pin
- **Size of PCB:** 58 x 30.2 mm
- **Weight (completely assembled):** 12g (with Overo SBC mounted: 16g)

5.4. Baseboard for COM Express SBC

5.4.1. Hardware Changes

The hardware changes applied to the baseboard for the COM Express SBC are described in the following list:

1. **HDMI Level Shifter:**
   As explained in section 4.1.3 an HDMI level shifter had to be integrated on the baseboard for correct operation of the HDMI interface. The HDMI level shifter STHDLS101 from ST Microelectronics was selected for this purpose. It was taken care during the layout process that the the HDMI signal tracks are as straight and as short as possible to fulfill the HDMI specifications.

2. **VGA:**
   A VGA port implemented on a 7-pin PicoBlade connector has been permanently integrated on the baseboard as a backup for the HDMI interface.

3. **USB**
   All 8 available USB ports have been routed to USB connectors (except for USB0 which is used for the USB-to-UART converter). USB1 is available on a 4-pin PicoBlade connector, USB2 / USB3 / USB4 have been routed to both USB Micro-A and PicoBlade

connectors and USB5 is connected to a USB Micro-A connector only. USB6 and USB7 are attached to USB A connectors. These two ports can be assigned to the second USB 2.0 controller of the Intel GS45 chipset in the BIOS to increase the throughput.

4. **LEDs:**
   Three LEDs have been added to GPIO pins for state display during flight.

5. **Interface to Gumstix Overo SBC:**
   The connector for one Gumstix Overo SBC has been removed to save edge space as the Overo platform has definitely been superseded.

6. **Power Supply:**
   A power diode STPS15L25D has been added between the 18.5V and the 11.1V power pads. This diode can be mounted instead of the TI PTN78020WAH DC/DC converter module. In this case a second 11.1V power source can be attached to the 18.5V pads. This way the SBC electronics can be powered from a laboratory power supply while the MAV is on the ground. Then for the flight a battery pack can be connected to the second pad pair while the system is operating. This increases the achievable flight duration. It is important that the battery pack is always attached to the 18.5V pads because only these pads are protected by the diode from a reverse current flowing into the power source. Only one diode has been implemented because of the large size of the power diode and because the supply voltage decreases by max. 0.35V over the diode. This diode has been particularly selected because of the low voltage drop.

7. **3.3V Power Rail:**
   The 3.3V power had been underpowered on the prototype of the COM Express baseboard due to using only a single TI TPS73633 [51] LDO for the whole board. The 3.3V power supply for the XBee wireless module and the external SATA device is now provided by a LD1117AS33TR high power LDO from ST Microelectronics with a maximum output current of 1A. The HDMI level shifter also has its own TI TPS73633 LDO for its 3.3V power supply.

8. **RTC Battery**
   An RTC battery implemented with a gold capacitor has been integrated on the baseboard. This battery keeps the real time clock (RTC) of the COM alive during power-off. Otherwise the RTC is reset whenever power is removed. This destroys the journal of journaling file systems like ext4. The consequence of this is that such file systems need to be repaired after every power-off state. Additionally the time stamps of recorded images during flights will not agree with the "real world" time, if the clock is not set correctly. This is not crucial but it improves the usability of the SBC electronics.

9. **IMU Mounting:**
   To create room on the PCB for the added functionality, the IMU cutout had to be removed. The IMU can now be mounted straight beneath the COM Express baseboard with spacers.

5.4.2. Final Specifications

The schematics of the revised baseboard for the COM Express SBC can be found in appendix A.4 and a connector description is given in appendix B.4. The final specifications are shown below:

5.4.2.1. Kontron COM Express Module Specifications

- **CPU:** Intel Core 2 Duo SL9400, 1.86GHz, 6MB L2 Cache
- **RAM:** 2GB DDR3, 1066MHz
- **Chipset:** Intel GS45
- **GPU:** Intel GMA X4500MHD
  - 10 Vector processors
  - Programmable shaders / GLSL / CL support
- **Size of PCB:** 95 x 95 mm
- **Weight:** 89g (incl. RAM)
- **Kontron Part Number:** 36004-0000-19-2

5.4.2.2. COM Express Baseboard Specifications

- **On-Board Power Supply:**
  - **12V / 72W** regulated power output generated by a TI PTN78020WAH switching DC/DC converter module complying with COM Express standard (14.5V to 36V battery necessary)
  - direct, slew-rate limited 11.1V power input from battery as an option (works with Kontron COM Express module but does not comply with standard)
  - **5V rail:** 30W (generated by a TI PTH08T230W DC/DC converter module)
  - **3.7V rail:** 22W (generated by a TI PTH08T230W DC/DC converter module)
- **Power consumption under full load (CPU and GPU) without peripherals:**
  - **Peak:** 30W
- **External Interfaces:**
  - 1x SATA-II receptacle with 3-pin PicoBlade power connector for SATA SSD
  - 1x HDMI mini (display)
  - 2x USB-A port
  - 4x USB on 4-pin PicoBlade (3x shared with USB micro-A ports)
  - 4x USB micro-A ports (3x shared with USB on 4-pin PicoBlade)
  - 2x GPIO connectors with 4 GPIO pins on each connector (8 GPIO connections in total)

  - 3x UART on 4-pin PicoBlade (emulated by a FTDI FT4232HL USB-to-UART converter [40] on USB port 0)
  - 1x I^2C on 4-pin PicoBlade
  - Interface to PIXHAWK IMU module (including power)
  - Servo Controller for up to 6 servos
  - Motor Controller interface for up to 4 digital I^2C Motor Controllers
  - XBee Pro 2 Wireless Module for 2.4GHz RF communication
- **Size of PCB:** 95 x 95 mm
- **Weight:**
  - bare PCB: 35g
  - assembled: 90g

5.5. Bus-Powered USB-to-UART Adapter

No hardware revision had to be done for this board since the prototype has already been fully operational and no other functionality had to be added.

5.5.1. Final Specifications

The schematics of the USB-to-UART adapter can be found in appendix A.5 and a connector description is given in appendix B.5. The final specifications are shown below:

- **Power Input:** powered from USB bus
- **Power Output:** min. 250mA @ 3.3V
- **Interfaces:**
  - 1x micro-B USB 2.0 slave port
  - 4x UART with 3.3V power supply on 4-pin PicoBlade connector

6. Summary & Outlook

6.1. Summary

This master thesis deals with the implementation of a complete flight electronics system for autonomous Micro Air Vehicles (MAVs) consisting of power supply, orientation sensing platform and on-board computer to execute sophisticated computer vision algorithms. Two different on-board computer systems have been designed, one for ultra light-weight, ultra low-power MAVs, e.g. a coaxial helicopter, the other one for slightly larger but fully autonomous MAVs, e.g. a quadrocopter. The orientation sensing platform (IMU) is the same for both use cases.

The ultra light-weight electronics are based on the previous PIXHAWK hardware and also comprise the same Gumstix Overo SBC. The electronics system was split up into three separate units (power supply, IMU and SBC electronics) to achieve more spatial flexibility for mounting the electronics on the MAV. Also the number of interfaces available for external peripherals has been greatly increased to allow for simple expansion of the functionality. While the functionality and flexibility of the flight electronics could be increased the overall system weight was decreased from 61g of the previous electronics to 46g.

The larger flight electronics system was designed to be capable of providing enough computational power to permit fully autonomous operation of the MAV by means of computer vision. It is based on an Intel Core 2 Duo SL9400 CPU integrated on a 95x95mm COM. The overall system weight of this highly performant on-board computer (including the orientation sensing platform) is only 250g. The power consumption of this system has been measured to stay below 30W under full load.

The orientation sensing platform, called IMU, is the same for both electronic systems. Its sensing capabilities include 3D linear acceleration, 3D angular velocity, 3D magnetic field, 1D absolute barometric pressure and temperature measurement. The sensor fusion is done on the platform by the embedded software running on the integrated ARM7 based microcontroller for real-time computation of the orientation angles. The attitude control of the MAV can also be done on the microcontroller.

Prototypes of all these platforms have been manufactured, assembled, programmed (if necessary) and tested. The test results have been used for the revision of the hardware design. The final hardware design is shown in appendix A and is currently being manufactured.

6.2. Outlook

A new mechanical quadrotor helicopter frame has been designed in parallel for the high performance flight electronics. This system will be used by the PIXHAWK project team in upcoming MAV competitions. It will also be used by the MAV research team of the Computer Science and Artificial Intelligence Laboratory (CSAIL) at the Massachusetts Institute of Technology.

Since this helicopter is a completely new design, the attitude controller has to be redeveloped from scratch. Real flight data has to be recorded to assess the characteristics of the new flight electronics under operational conditions.

The COM Express SBC electronics will not need a redesign in the near future to improve performance because it is based on the standardized COM Express modules. This means that new CPUs can be integrated into the electronics by simply purchasing a new COM Express module from a supplier. As far as the IMU is concerned, the sensor market is evolving rapidly. New smaller and more precise sensors have been announced already during this semester thesis, e.g.:

- **ST LSM303DLH [58]:** 3D linear accelerometer and 3D magnetometer in one package
- **Invensense ITG-3200 [59]:** digital 3D gyroscopic angular velocity sensor

Measurement data quality of these sensors would need to be evaluated in more detail. But if these sensors can adhere to their promises, an even smaller IMU could be designed. Anyway, it is only a question of time until a complete orientation sensing platform possibly already including sensor fusion algorithms will be integrated into one IC package. Until then, a redesign of the IMU platform will probably not be sensible because, beside the hardware development effort, a redesign also generates software and testing overhead.

Bibliography

[1] *PIXHAWK Wiki*, PIXHAWK, http://pixhawk.ethz.ch/.

[2] *Datasheet: SCA3100-D04 Three-Axis Digital Accelerometer*, Revision 1 ed., VTI Technologies, http://www.vti.fi/midcom-serveattachmentguid-de641aeef35911dd95216df8d3661e391e39/sca3100-d04_accelerometer_datasheet_82_688_00_d.pdf, 2009.

[3] *Datasheet: IDG-500 Dual-Axis Gyro*, Preliminary release ed., Invensense, http://invensense.com/mems/gyro/documents/PS-IDG-0500-00-06.pdf, 2009.

[4] *Datasheet: BMP085 Digital Pressure Sensor*, Revision 1.2 ed., Bosch Sensortec, http://www.bosch-sensortec.com/content/language1/downloads/BST-BMP085-DS000-05.pdf, 2009.

[5] *microETXexpress-PC Manual*, Revision 1.0 ed., Kontron, http://emea.kontron.com/_etc/scripts/download/getdownload.php?downloadId=OTMwMg==, 2010.

[6] *Datasheet: NB-BlackSilentFan XM1 / XM2*, Revision 1 ed., Noiseblocker, http://www.noiseblocker.de/de/documents/Datenblatt_BSF_XM_de.pdf, 2008.

[7] *Datasheet: XBee-PRO ZNet 2.5 OEM RF Modules*, Revision v1.x.4x ed., Digi International Inc., http://ftp1.digi.com/support/documentation/90000866_C.pdf, 2008.

[8] *Datasheet: PTH08T230W 6-A 4.5V-14V Input Non-Isolated Wide Output Adjustable Power Module*, Revision h ed., Texas Instruments, http://focus.ti.com/lit/ds/symlink/pth08t230w.pdf, 2008.

[9] *Datasheet: LDO06C C Class Non-Isolated DC/DC Converter*, Revision 2 ed., Emerson Network Power, http://www.powerconversion.com/assets/lf_ld06_1238545930_techref.pdf, 2008.

[10] *Point of Load Alliance (POLA)*, Texas Instruments, http://newscenter.ti.com/Blogs/newsroom/archive/2004/02/24/new-member-to-join-point-of-load-alliance-pola-sc04066.aspx.

[11] *Implementation of PWM in our software*, PIXHAWK, http://pixhawk.ethz.ch/software/embedded/flight/pwm.

[12] D. Gebre-Egziabher, R. C. Hayward, and J. D. Powell, "Design of multi-sensor attitude determination systems," *IEEE Transactions on Aerospace and Electronic Systems*, 2004.

[13] D. Gebre-Egziabher, G. H. Elkaim, J. D. Powell, and B. W. Parkinson, "A gyro-free quaternion-based attitude determination system suitable for implementation using low cost sensors," *Department of Aeronautics and Astronautics, Stanford University*, 2000.

[14] *Paparazzi Project Wiki*, ENAC University, http://paparazzi.enac.fr/wiki/Main_Page.

[15] *Datasheet: LPC214x User Manual*, Revision 1 ed., Philips Semiconductors, http://www.nxp.com/documents/user_manual/UM10139.pdf, 2005.

[16] *Datasheet: STM32F103xF, STM32F103xG User Manual*, Revision 2 ed., ST Microelectronics, http://www.st.com/stonline/products/literature/bd/17144.pdf, 2010.

[17] *Cortex-M3: Technical Reference Manual*, Revision g ed., ARM Limited, http://infocenter.arm.com/help/topic/com.arm.doc.ddi0337g/DDI0337G_cortex_m3_r2p0_trm.pdf, 2008.

[18] R. Winkler, H. Harms, and J. Schumm, "Garment-based measurement of human upper body orientation using miniature inertial sensing units," *Electronics Laboratory, ETH Zurich*, 2009.

[19] *Product Family Specification: SCA3000 Series*, Revision a.07 ed., VTI Technologies, http://www.vti.fi/midcom-serveattachmentguid-70faed9ef68211dd8a1f414702d2a290a290/sca3000_product_family_specification_8257300a.07.pdf, 2009.

[20] *Datasheet: ADXL345 Three-Axis Digital Accelerometer*, Revision a ed., Analog Devices, http://www.sparkfun.com/datasheets/Components/ADXL345.pdf, 2008.

[21] *Datasheet: MMA7455L Three-Axis Digital Accelerometer*, Revision 10 ed., Freescale, http://www.freescale.com/files/sensors/doc/data_sheet/MMA7455L.pdf, 2009.

[22] *Datasheet: LIS302DL MEMS Inertial Sensor*, Revision 4 ed., ST Microelectronics, http://www.st.com/stonline/products/literature/ds/12726.pdf, 2008.

[23] *Datasheet: LIS3LV02DQ MEMS Inertial Sensor*, Revision 1 ed., ST Microelectronics, http://www.st.com/stonline/products/literature/ds/11115.pdf, 2005.

[24] *Datasheet: ISZ-500 Single-Axis Z-Gyro*, Preliminary release ed., Invensense, http://invensense.com/mems/gyro/documents/PS-ISZ-0500-00-03.pdf, 2009.

[25] *Datasheet: 16-Bit, 4-Channel Serial Output Sampling ADC*, Revision 2 ed., Texas Instruments, http://focus.ti.com/lit/ds/symlink/ads8341.pdf, 2003.

[26] *Datasheet: MS2100 2-Axis Magnetic Sensor*, Revision 2 ed., PNI Sensor Corporation, http://www.pnicorp.com/files/MS2100%20H%20Datasheet_DOC%201014559%20R01_09-22-09.pdf, 2009.

[27] *Datasheet: 3-Axis Digital Compass IC HMC5843*, Revision 1 ed., Honeywell, http://www.ssec.honeywell.com/magnetic/datasheets/HMC5843.pdf, 2009.

[28] *Datasheet: AK8973 3-axis Electronic Compass*, Preliminary release ed., AKM, http://www.ic-on-line.cn/iol/datasheet/ak8973_4138699.pdf, 2007.

[29] *Datasheet: Integrated Compass Sensor HMC6052*, Revision a ed., Honeywell, http://www.ssec.honeywell.com/magnetic/datasheets/HMC6052.pdf, 2006.

[30] *Datasheet: SCP1000-D01 Digital Pressure Sensor*, Revision 1 ed., VTI Technology, http://www.vti.fi/midcom-serveattachmentguid-9cbae6a382efd245cb62354a54ff62c7/scp1000-d01_-d11_pressure_sensor_datasheet_28-08-2007.pdf, 2007.

[31] *Datasheet: Gumstix Overo Hardware Overview*, Gumstix, http://www.gumstix.net/Hardware/view/Motherboard-I/O/Overo-I/O-overview/112.html.

[32] *Datasheet: OMAP3530/25 Applications Processor*, Revision f ed., Texas Instruments, http://focus.ti.com/lit/ds/symlink/omap3530.pdf, 2009.

[33] *Datasheet: ADG3308 Low Voltage, 1.15 V to 5.5 V, 8-Channel Bidirectional Logic Level Translators*, Revision c ed., Analog Devices, http://www.analog.com/static/imported-files/data_sheets/ADG3308_3308-1.pdf, 2007.

[34] *Datasheet: ST2329A 2-bit dual supply level translator without direction control pin*, Revision 2 ed., ST Microelectronics, http://www.st.com/stonline/products/literature/ds/14907/st2329a.pdf, 2009.

[35] *Datasheet: PCA9306 Dual Bidirectional I2C Bus and SMBus Voltage-Level Translator*, Revision i ed., Texas Instruments, http://focus.ti.com/lit/ds/symlink/pca9306.pdf, 2008.

[36] *On-The-Go and Embedded Host Supplement to the USB Revision 2.0 Specification*, Revision 2.0 ed., USB Implementers Forum, http://www.usb.org/developers/onthego/USB_OTG_and_EH_2-0.pdf, 2009.

[37] *Datasheet: UPD720114 ECOUSB Series USB 2.0 Hub Controller*, Revision 1 ed., NEC Electronics, http://www.necel.com/nesdis/image/S17462EJ6V0DS00.pdf, 2005.

[38] *Datasheet: USB2514 USB 2.0 Hi-Speed Hub Controller*, Revision 1.1 ed., SMSC, http://www.smsc.com/media/Downloads_Public/Data_Sheets/251x.pdf, 2010.

[39] *Datasheet: ISP1520 USB Hi-Speed Hub Controller*, Revision 6 ed., ST Ericsson, http://www.stericsson.com/technical_documents/CD00222694.pdf, 2009.

[40] *Datasheet: FT4232H Quad High Speed USB to Multipurpose UART/MPSSE IC*, Revision 2.07 ed., Future Technology Devices International Limited, http://ftdichip.com/Documents/DataSheets/DS_FT4232H_V207.pdf, 2009.

[41] *Mobile-ITX Specification*, Revision 1 ed., VIA Technologies, http://www.via.com.tw/en/downloads/whitepapers/initiatives/spearhead/WP091201-ITXFormFactor.pdf, 2009.

[42] *SOMs/COMs with PCI Express*, Revision 1 ed., MEN Mikro Elektronik, http://www.men.de/docs-ext/products/pdf/comparecharts/hw_somcom_with_pciexpress.pdf, 2009.

[43] *Specifications - PC/104-PLUS*, PC/104 Embedded Consortium, http://www.pc104.org/pc104_plus_specs.php.

[44] *VIA Pico-ITX Form Factor*, Revision 1 ed., VIA Technologies, http://www.via.com.tw/en/downloads/whitepapers/initiatives/spearhead/pico-itx_form_factor.pdf, 2007.

[45] *COM Express Technology Concept*, Kontron, http://emea.kontron.com/_etc/scripts/download/getdownl

[46] *ETX Specification*, Revision 3.02 ed., ETX Industrial Group, http://emea.kontron.com/_etc/scripts/download/getdownload.php?downloadId=MTYxNQ==, 2007.

[47] *MSC CXC-GS45 Manual*, Revision 1.2 ed., MSC Vertriebs GmbH, http://www.msc-ge.com/de/4847-www/version/default/part/AttachmentData/data/MSC_CXC_GS45_UserManual.pdf, 2010.

[48] *COM Express Carrier Design Guide*, Revision 1.0 ed., PICMG, http://www.picmg.org/pdf/PICMG_COMDG_100.pdf, 2009.

[49] *Datasheet: PTN78020W 6-A Wide-Input Voltage Adjustable Switching Regulator*, Revision b ed., Texas Instruments, http://focus.ti.com/lit/ds/symlink/ptn78020w.pdf, 2008.

[50] *Datasheet: SUM110P04-05 P-Channel 40-V (D-S) MOSFET*, Revision b ed., Vishay Siliconix, http://www.vishay.com/docs/73493/73493.pdf, 2008.

[51] *Datasheet: TPS73633 Cap-Free, NMOS, 400mA Low-Dropout Regulator*, Revision s ed., Texas Instruments, http://focus.ti.com/lit/ds/symlink/tps73633.pdf, 2008.

[52] *Wasp UAV Documentation*, John Stowers, http://www.johnstowers.co.nz/files/wasp/index.html.

[53] *PIXHAWK IMU/flight Code Documentation*, PIXHAWK, http://pixhawk.ethz.ch/api/flight/.

[54] *Datasheet: 93AA86 16K Microwire Compatible Serial EEPROM*, Revision j ed., Microchip Technology, http://ww1.microchip.com/downloads/en/DeviceDoc/21797J.pdf, 2008.

[55] *Datasheet: 24FC256 256K I2C CMOS Serial EEPROM*, Revision p ed., Microchip Technology, http://ww1.microchip.com/downloads/en/DeviceDoc/21203P.pdf, 2007.

[56] *Benchmark Results*, PIXHAWK, http://pixhawk.ethz.ch/software/optimization/benchmark_results.

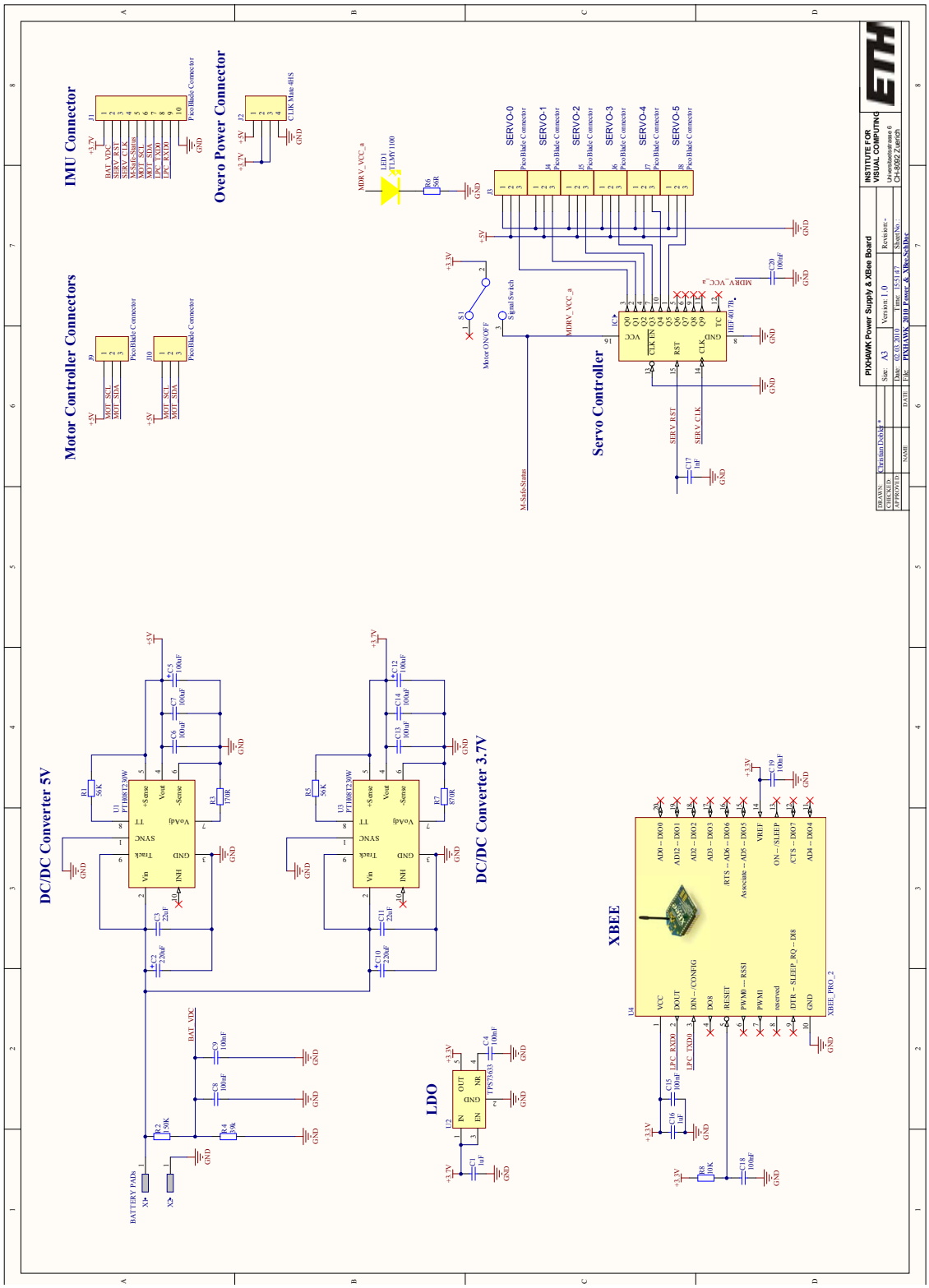
[57] *Datasheet: CY7C65640A TetraHub High Speed USB Hub Controller*, Revision j ed., Cypress Semiconductor, http://media.digikey.com/pdf/Data%20Sheets/Cypress%20PDFs/CY7C65640A.pdf, 2009.

[58] *Datasheet: LSM303DLH Sensor module: 3-axis accelerometer and 3-axis magnetometer*, Revision 1 ed., ST Microelectronics, http://www.st.com/stonline/products/literature/ds/16941.pdf, 2009.

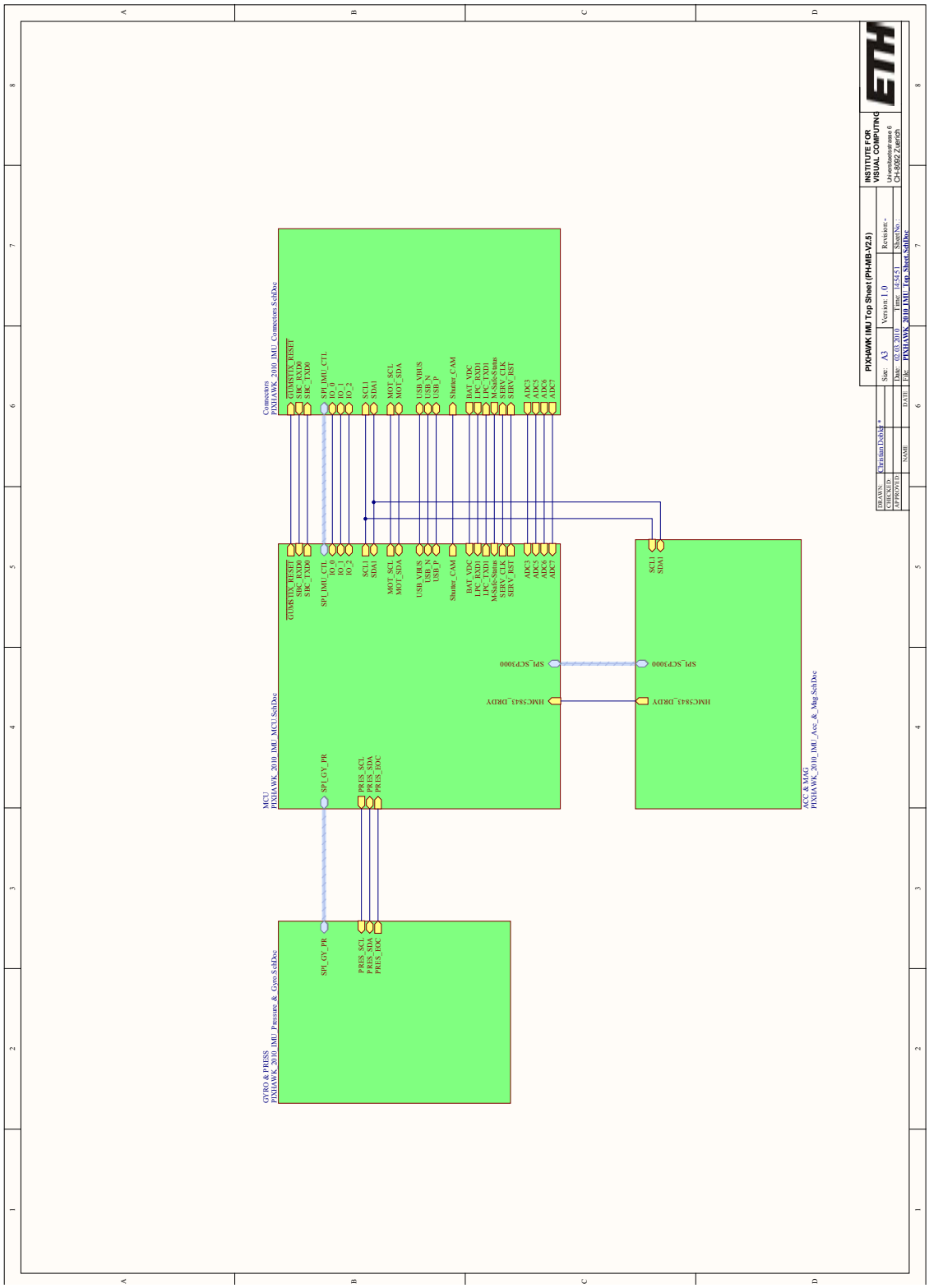
[59] *ITG-3200 Integrated Triple-Axis Digital-Output Gyroscope*, Invensense, http://invensense.com/mems/gyro/itg3200.html.

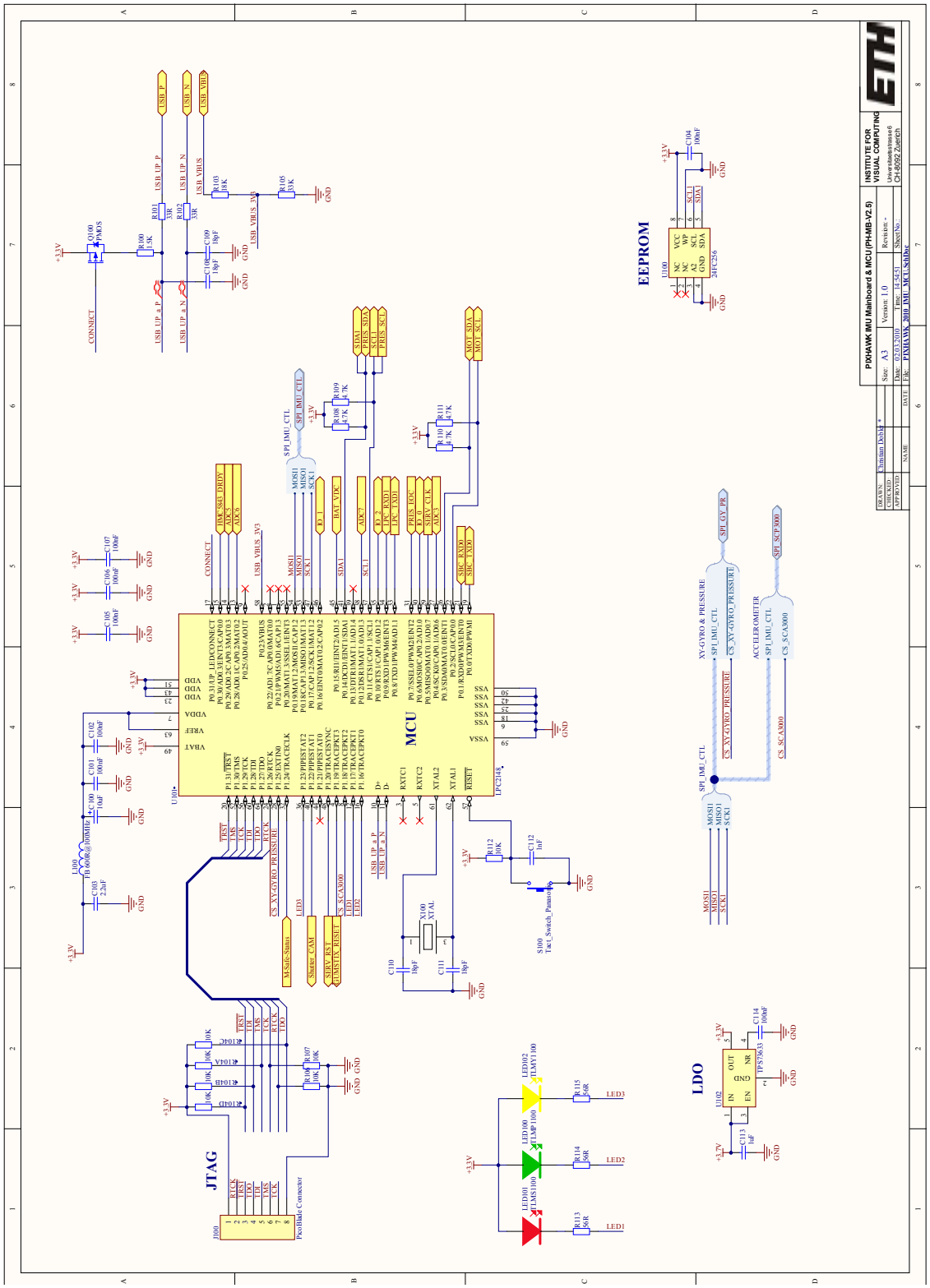
A. Final Schematics

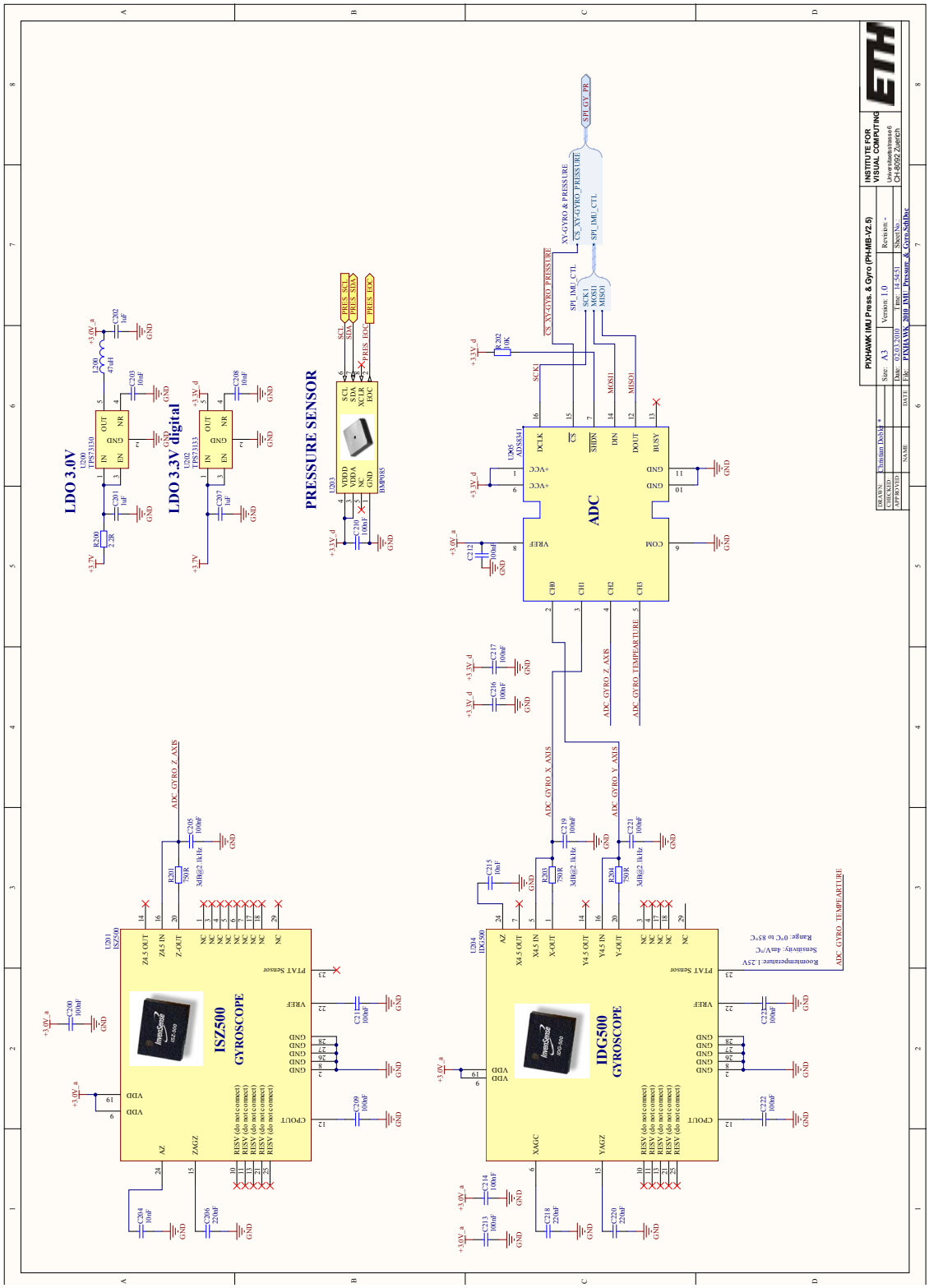
A.1. Power Supply

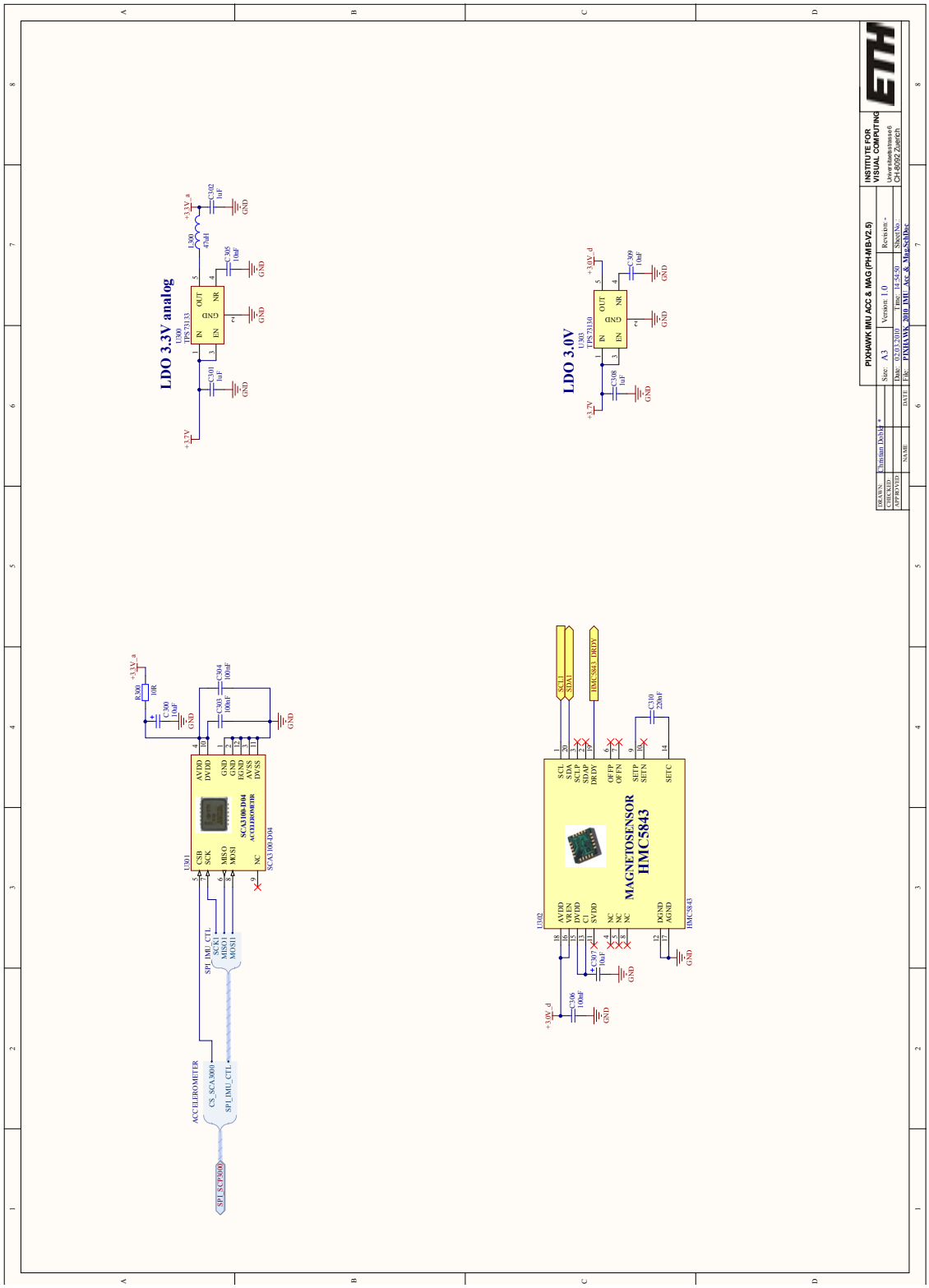


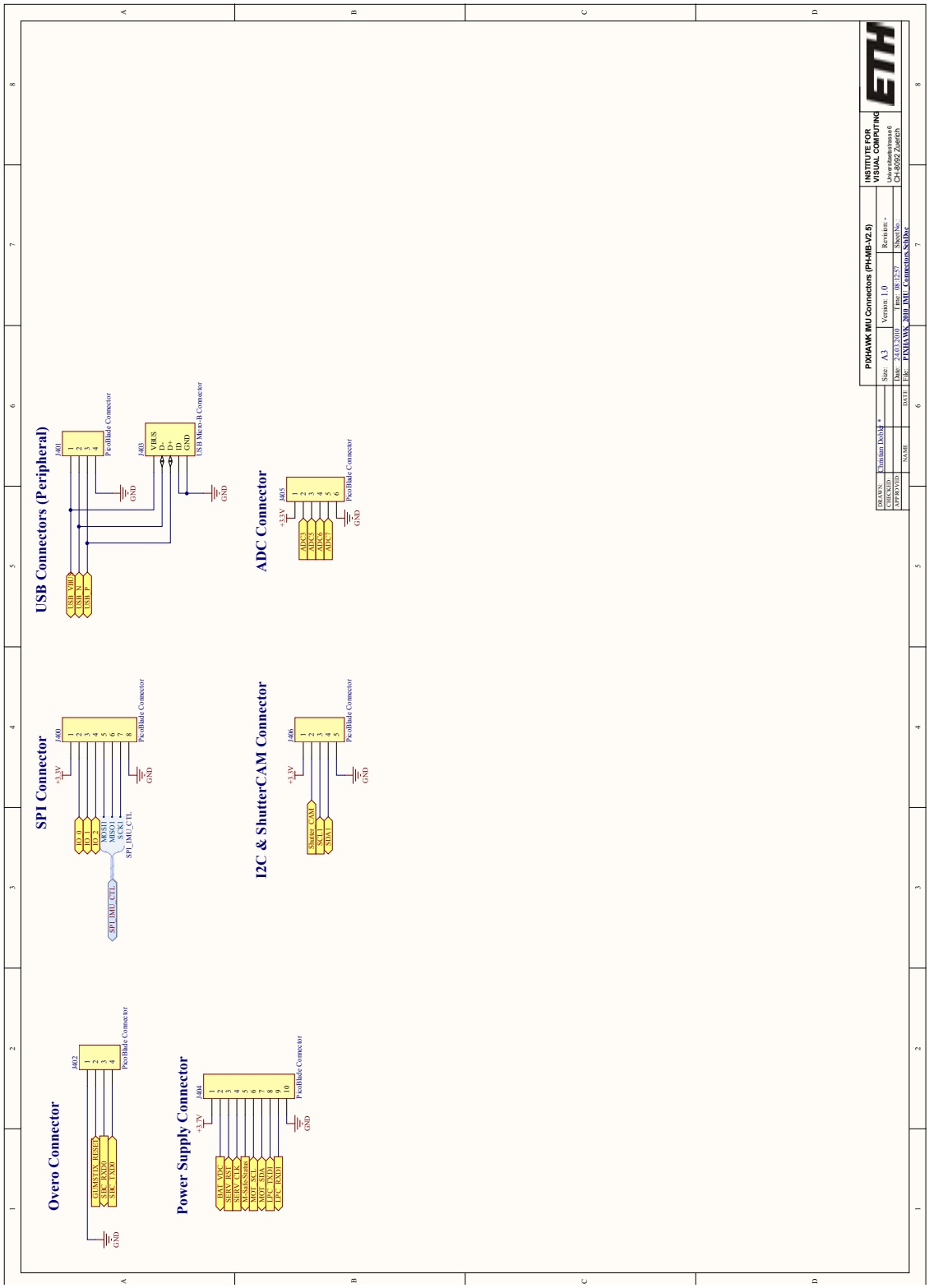
A.2. Inertial Measurement Unit



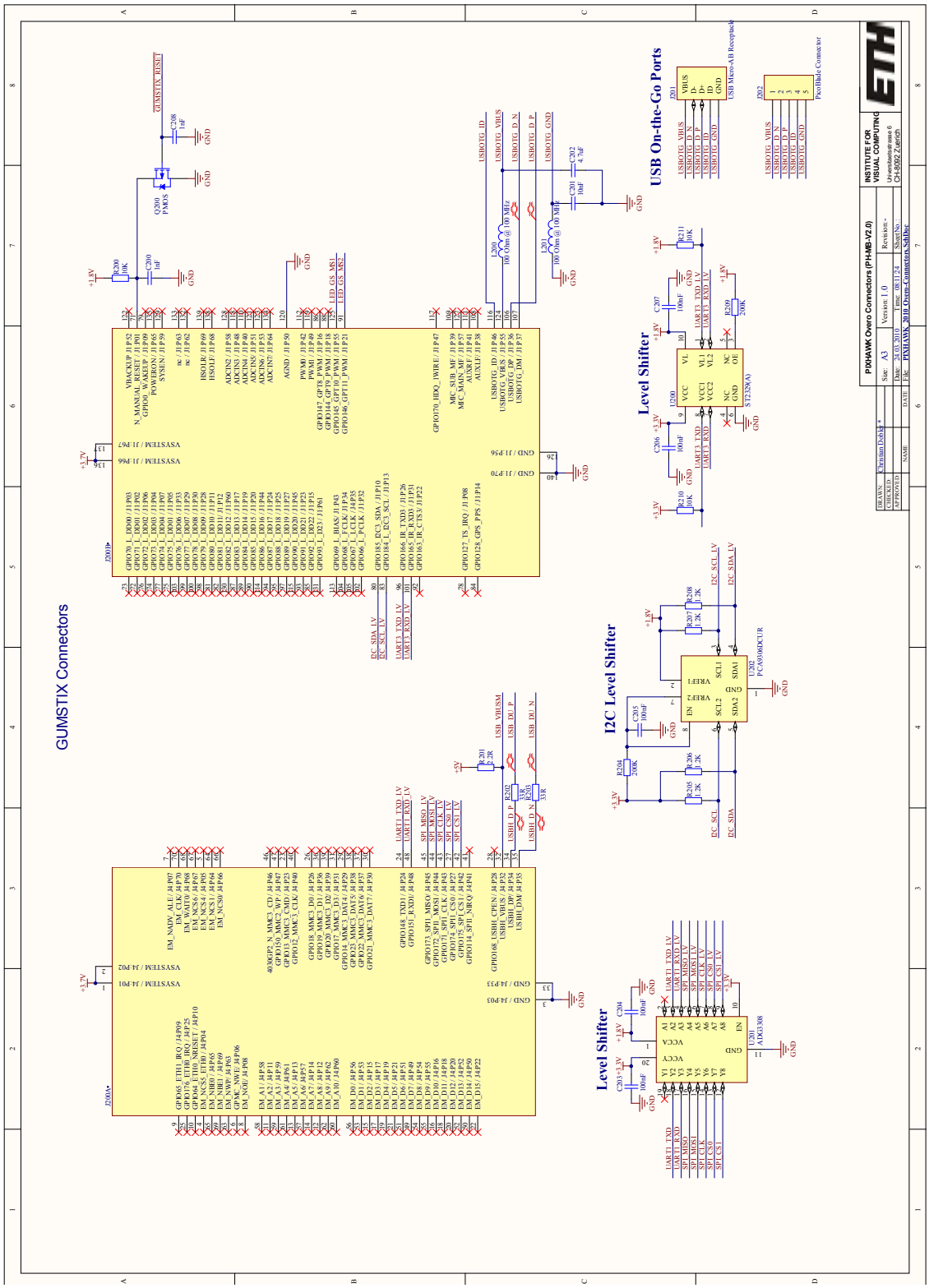

PIXHAWK IMU Mainboard & MCU (PH-MB-V2.5)
INSTITUTE FOR VISUAL COMPUTING
Size: A3
Version: 1.0
Revision: -
Date: 02.03.2010
Time: 14:54:51
Sheet No.:
File: PIXHAWK_2010_IMU_MCU.SchDoc
MCU
LPC2148
EEPROM
JTAG
LDO
CONNECT
ACCELEROMETER
XY-GYRO & PRESSURE
LED1
LED2
LED3

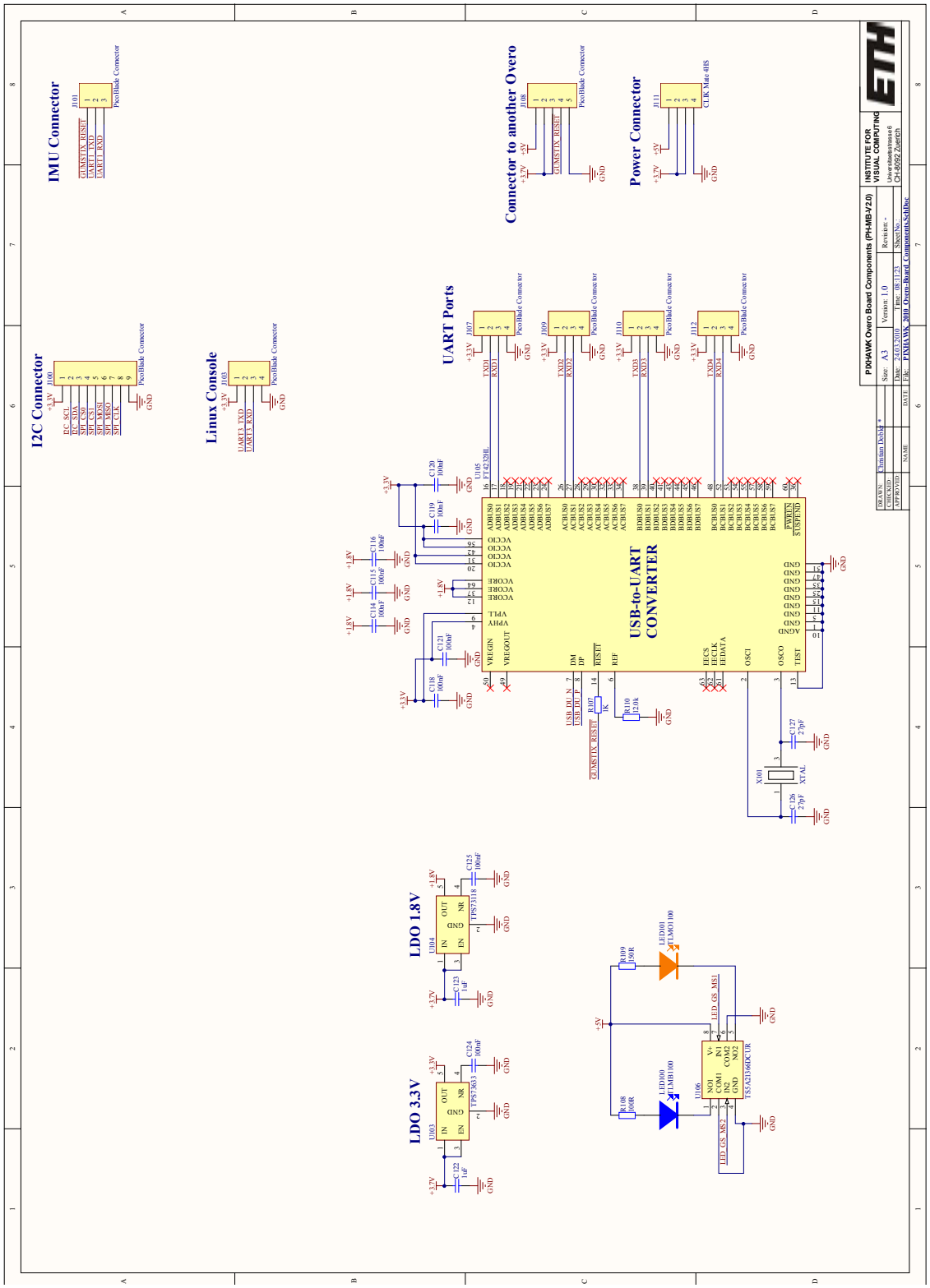

LDO 3.0V
LDO 3.3V digital
PRESSURE SENSOR
ADC
ISZ500 GYROSCOPE
IDG500 GYROSCOPE
PIXHAWK (IMU Press. & Gyro (PH-HB-V2.5)
INSTITUTE FOR VISUAL COMPUTING
Size: A3
Version: 1.0
Revision: -
Date: 02.03.2010
Time: 14:54:51
Sheet No.
File: PIXHAWK_2010_IMU_Pressure_&_Gyro.Schbrc
Thomas Dubler
CH-8092 Zuerich


LDO 3.3V analog
LDO 3.0V
MAGNETOSENSOR
HMC5843
ACCELEROMETER
PIXHAWK IMU ACC & MAG (PH-MB-V2.5)
INSTITUTE FOR VISUAL COMPUTING
Size: A3
Version: 1.0
Revision: -
Date: 02.03.2010
Time: 14:54:50
Sheet No.:
File: PIXHAWK_2010_IMU_Acc_&_Mag.SchDoc
Thomas Dubler
CH-8092 Zuerich

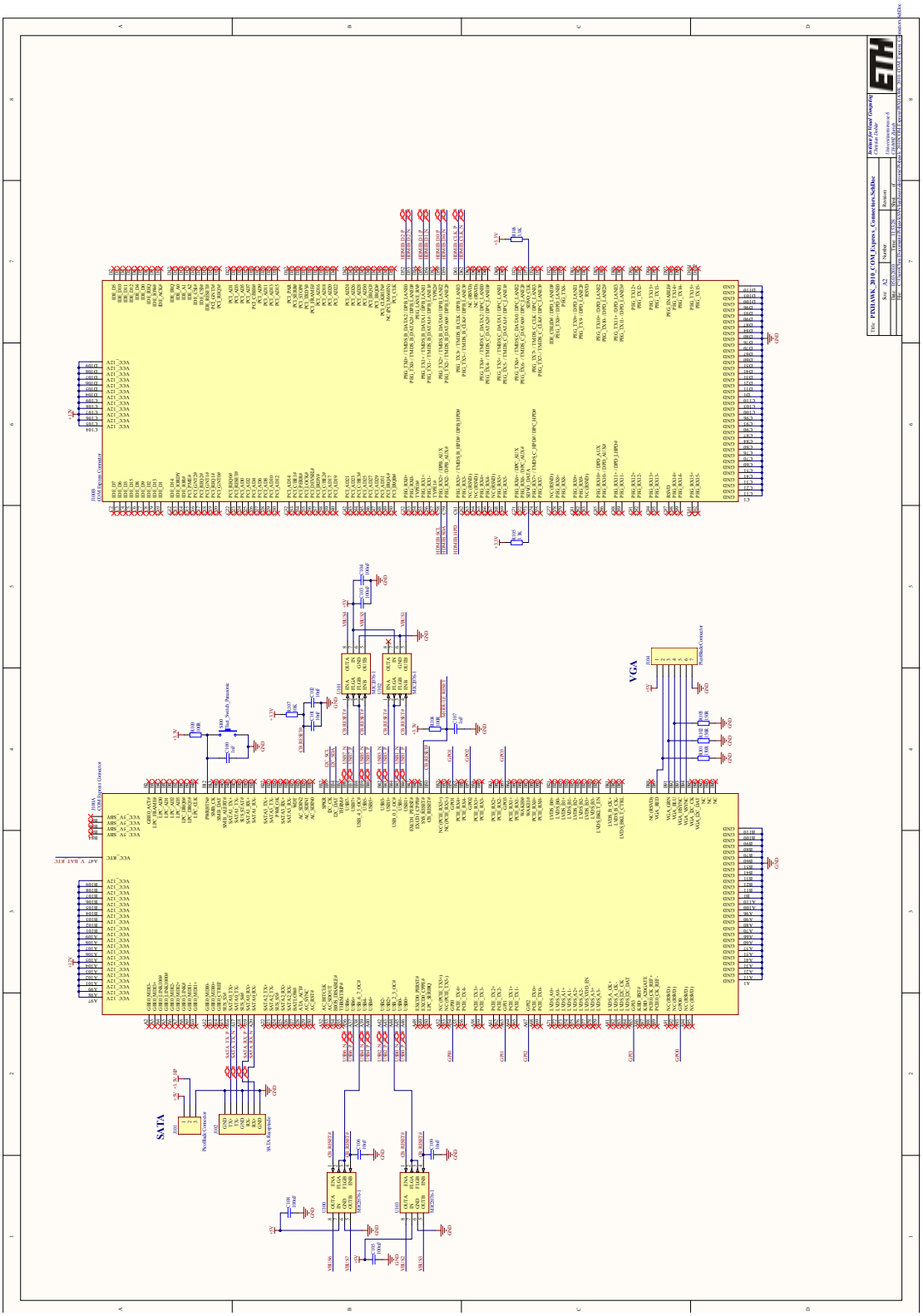

Overo Connector
J402
PicoBlade Connector
COM3/TX_RESET
SDC_RXD0
SDC_TXD0
GND
Power Supply Connector
+3.7V
J404
BAT_VDC
SERV_RST
SERV_CLK
Vbat_Switch
MOT_SCL
MOT_SDA
I2C_TXD
I2C_RXD
PicoBlade Connector
GND
SPI Connector
+3.3V
J400
IO 0
IO 1
AD 2
MOSI
MISO
SCK
SPI_IMU_CTL
SPI_IMU_CTL
PicoBlade Connector
GND
I2C & ShutterCAM Connector
+3.3V
J406
Shutter_CAM
SCL1
SDA1
PicoBlade Connector
GND
USB Connectors (Peripheral)
J401
USB_VBUS
USB_N
USB_P
PicoBlade Connector
GND
J403
VBUS
D-
D+
GND
USB Micro-B Connector
GND
ADC Connector
+3.3V
J405
AD 3
AD 5
AD 6
AD 7
PicoBlade Connector
GND
PIXHAWK IMU Connectors (PH-MB-V2.5)
Size: A3
Version: 1.0
Revision: -
Date: 24.03.2010
Time: 08:12:57
Sheet No.:
File: PIXHAWK_2010_IMU_Connectors.SchDoc
DRAWN
CHECKED
APPROVED
Christian Dubler *
NAME
DATE
INSTITUTE FOR
VISUAL COMPUTING
Universitätsstrasse 6
CH-8092 Zuerich
ETH

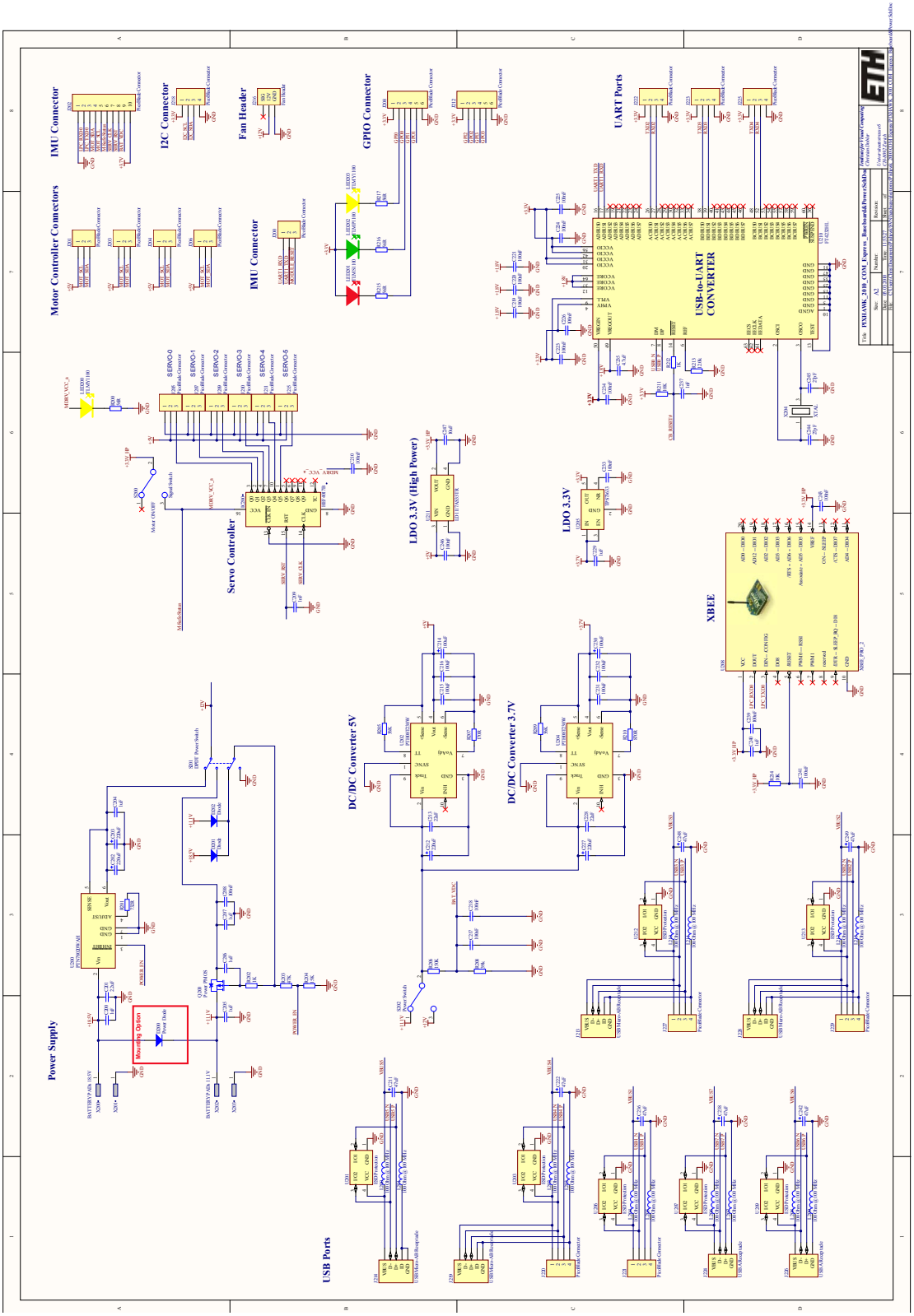
A.3. Baseboard for Gumstix Overo SBC

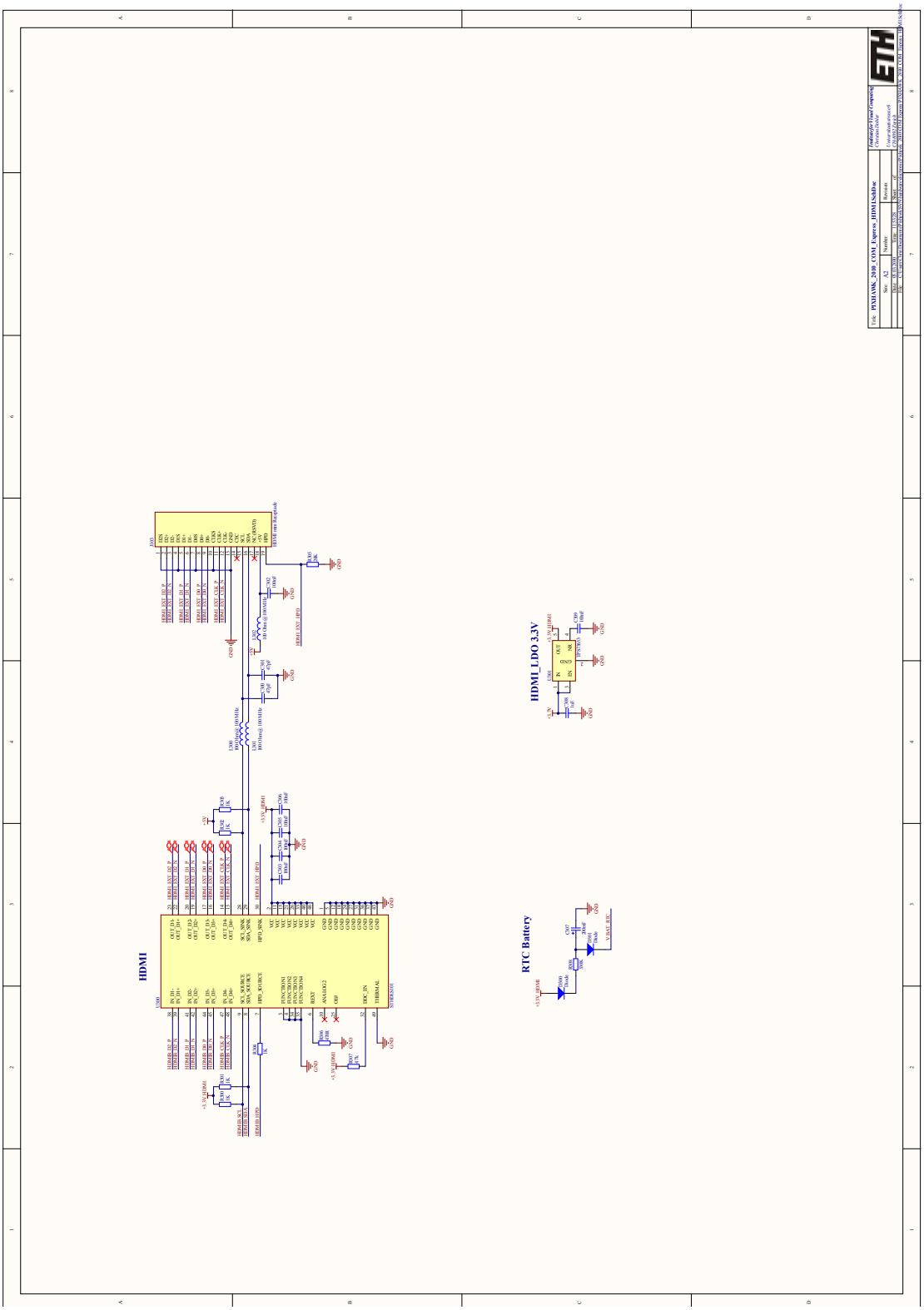



IMU Connector
I2C Connector
Linux Console
UART Ports
Connector to another Overo
Power Connector
USB-to-UART CONVERTER
LDO 1.8V
LDO 3.3V
PIXHAWK Overo Board Components (PH-HB-V2.0)
INSTITUTE FOR VISUAL COMPUTING
CH-8092 Zuerich

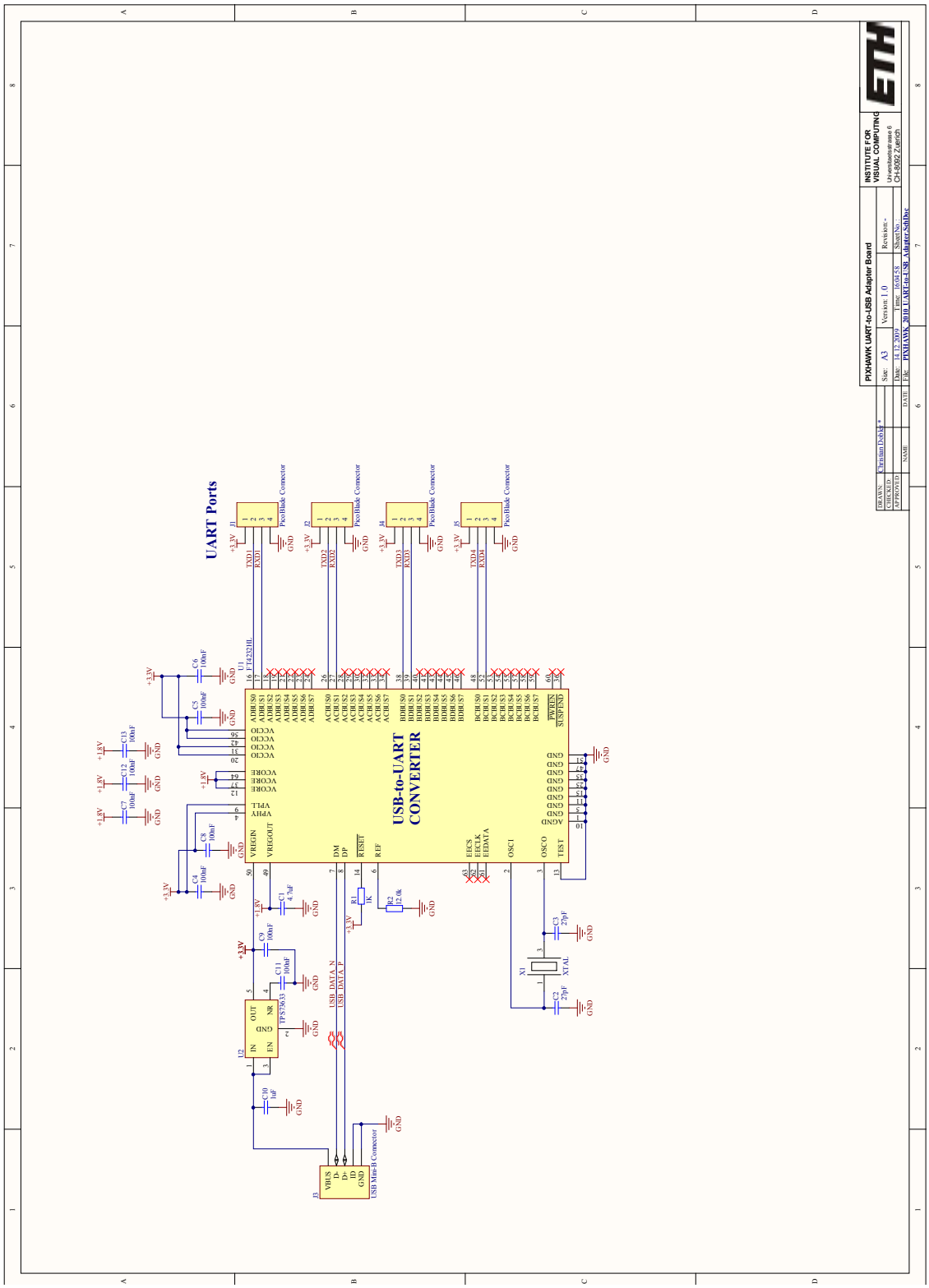
A.4. Baseboard for COM Express SBC




IMU Connector
I2C Connector
Fan Header
GPIO Connector
UART Ports
Motor Controller Connectors
IMU Connector
USB-to-UART CONVERTER
Servo Controller
LDO 3.3V (High Power)
LDO 3.3V
XBEE
DC/DC Converter 5V
DC/DC Converter 3.7V
Power Supply
USB Ports


HDMI
HDMI_LDO 3.3V
RTC Battery

A.5. Bus-Powered USB-to-UART Adapter



B. Description of Connectors

B.1. Power Supply

Interface to IMU Board: J1

Pin #	Name	Description
1	+3.7V	Power Output Pin (max. 1A)
2	BAT_VDC	Analog Battery Voltage Level Output
3	SERV_RST	Servo Controller reset signal
4	SERV_CLK	PWM Input to Servo Controller
5	M-Safe-Status	Servo Power State (on/off) output signal
6	MOT_SCL	I^2C clock line to digital Motor Controller (I^2C Bus 0)
7	MOT_SDA	I^2C data line to digital Motor Controller (I^2C Bus 0)
8	LPC_TXD0	UART 1 receive line to XBee wireless module from MCU (connect to LPC_TXD1 on J404 on IMU)
9	LPC_RXD0	UART 1 transmit line from XBee wireless module to MCU (connect to LPC_RXD1 on J404 on IMU)
10	GND	Ground Pin

Table B.1.: Connector Type: 10-pin Molex PicoBlade

Interface to Overo SBC Baseboard: J2

Pin #	Name	Description
1	+5V	5V Power Output Pin (max. 3A)
2	+3.7V	3.7V Power Output Pin (max. 6A combined)
3	+3.7V	3.7V Power Output Pin (max. 6A combined)
4	GND	Ground Pin

Table B.2.: Connector Type: 4-pin Molex CLIK Mate

Connectors for Servos: J3, J4, J5, J6, J7 & J8

Pin #	Name	Description
1	GND	Ground
2	+5V	5V Power Output Pin (max. 1A)
3	+Qx	PWM signal

Table B.3.: Connector Type: 3-pin Molex PicoBlade

Connectors for Digital Motor Controllers: J9 & J10

Pin #	Name	Description
1	+5V	5V Power Output Pin (max. 1A)
2	MOT_SCL	I^2C clock line to digital Motor Controller
3	MOT_SDA	I^2C data line to digital Motor Controller

Table B.4.: Connector Type: 3-pin Molex PicoBlade

B.2. Inertial Measurement Unit

Interface to Power Supply Board: J404

Pin #	Name	Description
1	+3.7V	Power Supply Pin (max. 1A)
2	BAT_VDC	Battery Voltage Input to AD1.5 to detect charging level
3	SERV_RST	Servo Controller reset signal
4	SERV_CLK	PWM Input to Servo Controller
5	M-Safe-Status	Servo Power State (on/off) input signal to P1.24
6	MOT_SCL	I^2C clock line to digital Motor Controller (I^2C Bus 0)
7	MOT_SDA	I^2C data line to digital Motor Controller (I^2C Bus 0)
8	LPC_TXD1	UART 1 transmit line to XBee wireless module
9	LPC_RXD1	UART 1 receive line from XBee wireless module
10	GND	Ground Pin

Table B.5.: Connector Type: 10-pin Molex PicoBlade

Interface to SBC Baseboard (Overo / COM Express Baseboard): J402

Pin #	Name	Description
1	GND	Ground Pin
2	GUMSTIX_RESET#	Reset signal to reset single board computer
3	SBC_RXD0	UART 0 receive line from single board computer (connect to UART1_TXD on SBC Baseboard)
4	SBC_TXD0	UART 0 transmit line to single board computer (connect to UART1_RXD on SBC Baseboard)

Table B.6.: Connector Type: 3-pin Molex PicoBlade

JTAG Connector: J100

Pin #	Name	Description
1	+3.3V	Power Output Pin (max. 250mA)
2	RTCK	see JTAG documentation on Wikipedia for details
3	TRST	
4	TDO	
5	TDI	
6	TMS	
7	TCK	
8	GND	Ground Pin

Table B.7.: Connector Type: 8-pin Molex PicoBlade

I^{2}C, Camera Shutter & Boot Pin Connector: J406

Pin #	Name	Description
1	+3.3V	Power Output Pin (max. 250mA)
2	Shutter_CAM	Shutter Signal for Cameras
3	SCL1	I^{2}C clock line for external I^{2}C devices (I^{2}C Bus 1)
4	SDA1	I^{2}C data line for external I^{2}C devices (I^{2}C Bus 1), Boot Pin: pull low during start-up to program MCU via UART0
5	GND	Ground Pin

Table B.8.: Connector Type: 5-pin Molex PicoBlade

SPI, PPM Input & I/O Connector: J400

Pin #	Name	Description
1	+3.3V	Power Output Pin (max. 250mA)
2	IO_0	PPM-Sum signal input, General Purpose I/O pin (connected to P0.6 on MCU)
3	IO_1	General Purpose I/O pin (connected to P0.16 on MCU)
4	IO_2	General Purpose I/O pin (connected to P0.10 on MCU)
5	MOSI1	SPI Master Out Slave In Bus line (SPI Bus 1)
6	MISO1	SPI Master In Slave Out Bus line (SPI Bus 1)
7	SCK1	SPI clock line (SPI Bus 1)
8	GND	Ground Pin

Table B.9.: Connector Type: 8-pin Molex PicoBlade

USB Connectors (slave ports): J401 & J403

Both receptacles are connected to the same USB interface. → **Only use one connector at a time!**

Pin #	Name	Description
1	USB_VBUS	USB VBUS (max. 500mA)
2	USB_N	negative differential USB data line (USB_DATA-)
3	USB_P	positive differential USB data line (USB_DATA+)
4	GND	Ground Pin

Table B.10.: J401: Connector Type: 4-pin Molex PicoBlade

Pin #	Name	Description
1	USB_VBUS	USB VBUS (max. 500mA)
2	USB_N	negative differential USB data line (USB_DATA-)
3	USB_P	positive differential USB data line (USB_DATA+)
4	ID	USB on-the-go (OTG) ID bus line → connected to GND
5	GND	Ground Pin

Table B.11.: J403: Connector Type: USB Micro-B Receptacle

ADC (Analog to Digital Converter) Connector: J405

Pin #	Name	Description
1	+3.3V	Power Output Pin (max. 250mA)
2	ADC3	analog input to AD0.6
3	ADC5	analog input to AD0.2
4	ADC6	analog input to AD0.1
5	ADC7	analog input to AD1.3
6	GND	Ground Pin

Table B.12.: J405: Connector Type: 6-pin Molex PicoBlade

B.3. Baseboard for Gumstix Overo SBC

Interface to Power Supply Board: J111

Pin #	Name	Description
1	+5V	5V Power Supply Pin (max. 3A)
2	+3.7V	3.7V Power Supply Pin (max. 6A combined)
3	+3.7V	3.7V Power Supply Pin (max. 6A combined)
4	GND	Ground Pin

Table B.13.: Connector Type: 4-pin Molex CLIK Mate

Interface to another Overo Board: J108

Pin #	Name	Description
1	+5V	5V Power Supply Pin (max. 1A)
2	+3.7V	3.7V Power Supply Pin (max. 2A combined)
3	+3.7V	3.7V Power Supply Pin (max. 2A combined)
4	GUMSTIX_RESET#	forwarded SBC reset signal from MCU on IMU
5	GND	Ground Pin

Table B.14.: Connector Type: 5-pin Molex PicoBlade

Interface to MCU on IMU Board: J101

Pin #	Name	Description
1	GUMSTIX_RESET#	Reset signal to reset single board computer
2	UART1_TXD	UART 1 transmit line to MCU on IMU board (connect to SBC_RXD0 on IMU)
3	UART1_RXD	UART 1 receive line from MCU on IMU board (connect to SBC_TXD0 on IMU)

Table B.15.: Connector Type: 3-pin Molex PicoBlade

Linux Console Interface: J103

Pin #	Name	Description
1	+3.3V	Power Output (max. 200mA)
2	UART3_TXD	UART 3 transmit line:= serial Linux console output
3	UART3_RXD	UART 3 receive line:= serial Linux console input
4	GND	Ground Pin

Table B.16.: Connector Type: 4-pin Molex PicoBlade

UART Ports: J107, J109, J110, J112

Pin #	Name	Description
1	+3.3V	Power Output (max. 200mA)
2	TXDx	UART transmit line to external UART device
3	RXDx	UART receive line from external UART device
4	GND	Ground Pin

Table B.17.: Connector Type: 4-pin Molex PicoBlade

I^2C & SPI Connector: J100

Pin #	Name	Description
1	+3.3V	Power Output Pin (max. 200mA)
2	I2C_SCL	I^2C clock line for external I^2C devices
3	I2C_SDA	I^2C data line for external I^2C devices
4	SPI_CS0	SPI Chip Select line 0
5	SPI_CS1	SPI Chip Select line 1
6	SPI_MOSI	SPI Master Out Slave In Bus line
7	SPI_MISO	SPI Master In Slave Out Bus line
8	SPI_CLK	SPI clock line
9	GND	Ground Pin

Table B.18.: Connector Type: 9-pin Molex PicoBlade

USB On-The-Go Port (USB_OTG): J201 & J202

Both receptacles are connected to the same USB interface. → **Only use one connector at a time!**

Pin #	Name	Description
1	USBOTG_VBUS	USB OTG VBUS (max. 500mA)
2	OTG_D-	negative differential USB data line (USB_DATA-)
3	OTG_D+	positive differential USB data line (USB_DATA+)
4	GND	Ground Pin

Table B.19.: J202: Connector Type: 4-pin Molex PicoBlade

Pin #	Name	Description
1	USBOTG_VBUS	USB OTG VBUS (max. 500mA)
2	OTG_D-	negative differential USB data line (USB_DATA-)
3	OTG_D+	positive differential USB data line (USB_DATA+)
4	ID	USB on-the-go (OTG) ID bus line
5	GND	Ground Pin

Table B.20.: J201: Connector Type: USB Micro-AB Receptacle

B.4. Baseboard for COM Express SBC

B.4.1. Interface to IMU Board

Communication Interface of the Baseboard Components with MCU on IMU: J202

Pin #	Name	Description
1	GND	Ground Pin
2	LPC_RXD0	UART 1 transmit line from XBee wireless module to MCU (connect to LPC_RXD1 on J404 on IMU)
3	LPC_TXD0	UART 1 receive line to XBee wireless module from MCU (connect to LPC_TXD1 on J404 on IMU)
4	MOT_SDA	I^2C data line to digital Motor Controller (I^2C Bus 0)
5	MOT_SCL	I^2C clock line to digital Motor Controller (I^2C Bus 0)
6	M-Safe-Status	Servo Power State (on/off) output signal
7	SERV_CLK	PWM Input to Servo Controller
8	SERV_RST	Servo Controller reset signal
9	BAT_VDC	Analog Battery Voltage Level Output
10	+3.7V	Power Output Pin (max. 1A)

Table B.21.: Connector Type: 10-pin Molex PicoBlade

Communication Interface of COM Express Module with MCU on IMU: J200

Pin #	Name	Description
1	UART1_RXD	UART 1 receive line from MCU on IMU board (connect to SBC_TXD0 on IMU)
2	UART1_TXD	UART 1 transmit line to MCU on IMU board (connect to SBC_RXD0 on IMU)
3	MODULE_RESET#	SBC reset signal issued by MCU on IMU board (connect to GUMSTIX_RESET on IMU)

Table B.22.: Connector Type: 3-pin Molex PicoBlade

B.4.2. Digital Data Interfaces

UART Ports: J222, J223 & J225

These UART interfaces are established by an FTDI FT4232HL USB-to-UART converter (U210), which is attached to USB Port 0 of the COM Express module.

Pin #	Name	Description
1	+3.3V	Power Output (max. 250mA)
2	TXDx	UART transmit line to external UART device
3	RXDx	UART receive line from external UART device
4	GND	Ground Pin

Table B.23.: Connector Type: 4-pin Molex PicoBlade

I^2C Connector: J218

Pin #	Name	Description
1	+3.3V	Power Output Pin (max. 250mA)
2	I2C_SCL	I^2C clock line for external I^2C devices
3	I2C_SDA	I^2C data line for external I^2C devices
4	GND	Ground Pin

Table B.24.: Connector Type: 4-pin Molex PicoBlade

USB Host Ports: J213, J214, 219, J220, J221, J224, J226, J227, J228 & J229

Receptacles *J219 and J220* are both connected to USB port 4. → **Only use one connector at a time!**
Receptacles *J213 and J227* are both connected to USB port 3. → **Only use one connector at a time!**
Receptacles *J228 and J229* are both connected to USB port 2. → **Only use one connector at a time!**

Pin #	Name	Description
1	VBUSx	USB OTG VBUS (max. 500mA)
2	USBx_N	negative differential USB data line (USB_DATA-)
3	USBx_P	positive differential USB data line (USB_DATA+)
4	GND	Ground Pin

Table B.25.: J220, J221, J227 & J229: Connector Type: 4-pin Molex PicoBlade

Pin #	Name	Description
1	USB_VBUS	USB OTG VBUS (max. 500mA)
2	D-	negative differential USB data line (USB_DATA-)
3	D+	positive differential USB data line (USB_DATA+)
4	ID	USB on-the-go (OTG) ID bus line → connected to GND
5	GND	Ground Pin

Table B.26.: J213, J214, J219 & J228: Connector Type: USB Micro-A Receptacle

Pin #	Name	Description
1	USB_VBUS	USB OTG VBUS (max. 500mA)
2	D-	negative differential USB data line (USB_DATA-)
3	D+	positive differential USB data line (USB_DATA+)
4	GND	Ground Pin

Table B.27.: J224 & J226: Connector Type: USB A Receptacle

GPIO Connectors: J208 & J212

Pin #	Name	Description
1	+3.3V	Power Output Pin (max. 250mA)
2	GPIOa	GPIO Pin a
3	GPIOb	GPIO Pin b
4	GPIOc	GPIO Pin c
5	GPIOd	GPIO Pin d
6	GND	Ground Pin

Table B.28.: Connector Type: 6-pin Molex PicoBlade

VGA: J208 & J212

Pin #	Name	Description
1	+5V	Power Pin (Pin 9 on VGA D-Sub)
2	VGA_RED	red color channel (Pin 1 on VGA D-Sub)
3	VGA_GRN	green color channel (Pin 2 on VGA D-Sub)
4	VGA_BLU	blue color channel (Pin 3 on VGA D-Sub)
5	VGA_HSYNC	horizontal synchronization (Pin 13 on VGA D-Sub)
6	VGA_VSYNC	vertical synchronization (Pin 14 on VGA D-Sub)
7	GND	Ground Pin (Pins 5,6,7,8 & 10 on VGA D-Sub)

Table B.29.: Connector Type: 7-pin Molex PicoBlade

HDMI (HDMI mini): J103

See HDMI specification for more detail.

SATA-II Receptacle (data): J102

See SATA specification for more detail.

B.4.3. Various Interfaces

SATA Power Connector: J101

Pin #	Name	Description
1	+5V	5V Power Output Pin (max. 1A)
2	+3.3V_HP	3.3V Power Output Pin (max. 700mA)
3	GND	Ground Pin

Table B.30.: Connector Type: 3-pin Molex PicoBlade

Connectors for Servos: J201, J207, J209, J210, J211 & J215

Pin #	Name	Description
1	GND	Ground
2	+5V	5V Power Output Pin (max. 1A)
3	+Qx	PWM signal

Table B.31.: Connector Type: 3-pin Molex PicoBlade

Connectors for Digital Motor Controllers: J201, J203, J204 & J206

Pin #	Name	Description
1	+5V	5V Power Output Pin (max. 1A)
2	MOT_SCL	I^2C clock line to digital Motor Controller
3	MOT_SDA	I^2C data line to digital Motor Controller

Table B.32.: Connector Type: 3-pin Molex PicoBlade

Standard Computer Fan Header: J216

Pin #	Name	Description
1	SIG	not connected
2	+12V	12V Power Output Pin
3	GND	Ground Pin

Table B.33.: Connector Type: 3-pin Molex Fan Header

B.5. Bus-Powered USB-to-UART Adapter

USB Connector (slave port): J3

Pin #	Name	Description
1	USB_VBUS	USB VBUS (max. 500mA)
2	USB_N	negative differential USB data line (USB_DATA-)
3	USB_P	positive differential USB data line (USB_DATA+)
4	ID	USB on-the-go (OTG) ID bus line → connected to GND
5	GND	Ground Pin

Table B.34.: Connector Type: USB Mini-B Receptacle

UART Ports: J1, J2, J4 & J5

Pin #	Name	Description
1	+3.3V	Power Output (min. 250mA)
2	TXDx	UART transmit line to external UART device
3	RXDx	UART receive line from external UART device
4	GND	Ground Pin

Table B.35.: Connector Type: 4-pin Molex PicoBlade

C. C Program Code

C.1. Main Program File (main.c)

```
/*=====================================================================

PIXHAWK Micro Air Vehicle Flying Robotics Toolkit

(c) 2009, 2010 PIXHAWK PROJECT  <http://pixhawk.ethz.ch>

This file is part of the PIXHAWK project

    PIXHAWK is free software: you can redistribute it and/or modify
    it under the terms of the GNU General Public License as published by
    the Free Software Foundation, either version 3 of the License, or
    (at your option) any later version.

    PIXHAWK is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
    GNU General Public License for more details.

    You should have received a copy of the GNU General Public License
    along with PIXHAWK. If not, see <http://www.gnu.org/licenses/>.

=====================================================================*/

/**
 * @file the main file
 *   @author Martin Rutschmann
 *   @author Dominik Honegger
 *   @author Tobias Naegeli
 *   @author Joseph Kruijen
 *   @author Lorenz Meier
 *   @author Christian Dobler
 *
 *  Please refer to
 *
 *  http://pixhawk.ethz.ch/software/embedded/flight/
 *  and
 *  http://pixhawk.ethz.ch/api/flight/
 *
 *  For the full documentation of this code.
 *
 */

// Select the board

// select the CPU
#include "inttypes.h"
```

```
#include "mcu_init.h"
#include "conf.h"
// Include comm
#include "comm.h"
#include "mavlink.h"
// Include globals
#include "global_data.h"
#include "sensors.h"
#include "calibration.h"

#include "adc.h"
//#include "buzzer.h"
//#include "cam_trigger.h"
#include "led.h"
#include "ppm.h"
#include "pwm.h"
#include "sys_time.h"
#include "uart.h"
#include "dac.h"

#include "watchdog.h"
#include "control.h"

#include "compass.h"
#include "acceleration.h"
#include "gyros.h"
#include "motors.h"
#include "servos.h"
#include "radio_control.h"
#include "battery.h"
#include "sca3100.h"

#include "global_pos.h"
#include "communication.h"

#include "simple_altitude_moving_average.h"
#include "attitude_compl_euler.h"
#include "altitude_kalman.h"
#include "lookup_sin_cos.h"
#include "least_square.h"

typedef enum
{
  COUNTER1 = 0,
  COUNTER2,
  COUNTER3,
  COUNTER4,
  COUNTER5,
  NUM_OF_COUNTERS
} counter_id_t;


void main_init (void);
void main_loop(void);
static inline void debug(void);
static inline void house_keeping(void);
static inline void house_keeping_v2(void);
```

```c
static inline uint8_t ms_run_every(uint32_t ms, counter_id_t counter_id);

uint32_t next_exec_time[NUM_OF_COUNTERS];



/**
 * @brief Initialize the whole system
 *
 * All functions that need to be called before the first mainloop iteration
 * should be placed here.
 */
void main_init (void){

        //uint8_t buffer[200];

        // Set mode to locked
        global_data.mode = MAV_MODE_UNINIT;

        // Lowlevel init
        hw_init();
        enableIRQ();
        led_init();
        sys_time_init();
        sys_time_periodic_init();
        sys_time_clock_init();
        ppm_init();
        pwm_init();

        // Read parameters from EEPROM
        // TODO This will be read from EEPROM

        global_data.param[PARAM_SYSTEM_ID] = 1;
        strcpy(global_data.param_name[PARAM_SYSTEM_ID], "SYSTEM_ID");

        global_data.param[PARAM_COMPONENT_ID] = 200;
        strcpy(global_data.param_name[PARAM_COMPONENT_ID], "COMPONENT_ID");

        global_data.param[PARAM_SYSTEM_TYPE] = MAV_COAXIAL;
        strcpy(global_data.param_name[PARAM_SYSTEM_TYPE], "SYSTEM_TYPE");

        global_data.param[PARAM_SW_VERSION] = 2000;
        strcpy(global_data.param_name[PARAM_SW_VERSION], "SW_VERSION");

        global_data.param[PARAM_PPM_SAFETY_SWITCH_CHANNEL] = 5;
        strcpy(global_data.param_name[PARAM_PPM_SAFETY_SWITCH_CHANNEL], "
            RC_SAFETY_CHAN");

        global_data.param[PARAM_UART0_BAUD] = 115200;
        strcpy(global_data.param_name[PARAM_UART0_BAUD], "UART0_BAUD");

        global_data.param[PARAM_UART1_BAUD] = 57600;
        strcpy(global_data.param_name[PARAM_UART1_BAUD], "UART1_BAUD");

        // Comm init
        comm_init(COMM_0);
        comm_init(COMM_1);
```

```
        uint8_t counter_id;
        for(counter_id=0; counter_id<NUM_OF_COUNTERS; counter_id++) {
                next_exec_time[counter_id] = 0;
        }

        // Lowlevel services are now initialized
        // Send boot message
        message_boot_send(COMM_0, global_data.param[PARAM_SW_VERSION]);
        //led_toggle(LED_RED);
        message_boot_send(COMM_1, global_data.param[PARAM_SW_VERSION]);

        // Sensor init
        sensors_init();

        #ifdef BOARD_PIXHAWK_V100
        motors_init();
        #endif
        servos_init();

        control_init();
//      buzzer_init();

        // Calibration starts (this can take a few seconds)
        led_on(LED_GREEN);
        led_on(LED_RED);
        message_send_state(COMM_0, MAV_STATE_CALIBRATING);
        message_send_state(COMM_1, MAV_STATE_CALIBRATING);

        //auto_calibration();

        // Calibration stopped
        led_off(LED_RED);

        // System is now in standby
        message_send_state(COMM_0, MAV_STATE_STANDBY);
        message_send_state(COMM_1, MAV_STATE_STANDBY);
}


/**
 * @brief This is the main loop
 *
 * It will be executed at maximum MCU speed (60 Mhz)
 */
void main_loop(void) {
//      static int z = 0;
//      IO0DIR |= 1<<21;
        //uint8_t buffer[200];

        #if defined(BOARD_PIXHAWK_V100) || defined(BOARD_TWOG_BOOZ)
        while (sys_time_periodic() != 1);
        unsigned int cycle10 = 0;
        #endif


        #ifdef IMU_PIXHAWK_V200
```

```
#endif

while (1)
{
        #if defined(IMU_PIXHAWK_V200) || defined(IMU_PIXHAWK_V210)

        communication_receive();

        if(ms_run_every(1000, COUNTER5)) {
                communication_send_1hz();
        }

        if(ms_run_every(100, COUNTER4)) {
                communication_send_10hz();
                led_toggle(LED_RED);
        }

        if(ms_run_every(20, COUNTER3)) {
                //sensors_read_out_50hz();
                complimentary_filter_predict_rad(global_data.accel_raw,
                     global_data.gyros_raw,global_data.magnet_raw,&
                    global_data.attitude,&global_data.gyros_si);
                control();
                communication_send_mainloop_rate();
                house_keeping_v2();
        }

        if(ms_run_every(10, COUNTER2)) {
                sensors_read_out_100hz();
        }

        if(ms_run_every(5, COUNTER1)) {
                sensors_read_out_200hz();
        }

        #endif


        #if defined(BOARD_PIXHAWK_V100) || defined(BOARD_TWOG_BOOZ)

        while (sys_time_periodic() != 1);

        if(cycle10 < 10) cycle10++;
        else {
                led_toggle(LED_RED);
                cycle10 = 0;
        }

        sensors_read_out();


        complimentary_filter_predict_rad(global_data.accel_raw,
            global_data.gyros_raw,global_data.magnet_raw,&global_data.
            attitude,&global_data.gyros_si);

        control();
```

```
                communication();

                house_keeping();

//              calibration();
//              debug();
//              static float theta[2] = {0,0};
//              static float p[4] = {1E4,0,0,1E4};
//
//              static int start=0;
//
//              if(start<50*5){
//                      least_square(global_data.pressure_raw, global_data.
    supersonic_raw,theta,p);
//                      start++;
//              }

//              static float Ts = 0.02f;
//              static float sigmaAcc = 0.1f;
//              static float sigmaPos = 0.05f;
//              static float gammaAcc = 1.0f;
//              static float gammaBiasAcc = 0.1f;
//
//              static float pkk_1[9] = {1000.0f,0.0f,0.0f,0.0f,1000.0f,0.0f
    ,0.0f,0.0f,1000.0f};
//              static float state_pred[3] = {0.0f, 0.0f, 0.0f};
//              float abs = ((float)global_data.supersonic_raw)/1000.0f;
//              float accel = -((float)global_data.accel_raw.z)/1333.0f*9.81f;
//              float bias_acc;
//              float speed_estimated;
//              float position_estimated;
//              altitude_kalman(abs,accel,Ts,sigmaAcc,sigmaPos,gammaAcc,
    gammaBiasAcc,pkk_1,state_pred,&speed_estimated,&bias_acc,&position_estimated
    );


//#define MAG_NEUTRALX -56
//#define MAG_NEUTRALY -17
//#define MAG_SCALEX 0.0208333333f
//#define MAG_SCALEY 0.0196078431f
//#define DIP_ANGLE 1.0472f
//
//              float phi = global_data.attitude.x;
//              float theta = global_data.attitude.y;
//
//              float_vect3 mag;
//          mag.x = (global_data.magnet_raw.x-MAG_NEUTRALX)*MAG_SCALEX;
//          mag.y = (global_data.magnet_raw.y-MAG_NEUTRALY)*MAG_SCALEY;
//              mag.z=(sin(DIP_ANGLE)+mag.x*sin(theta)-mag.y*cos(theta)*sin(phi
    ))/(cos(theta)*cos(phi));
//              static int k=1;
//              if(k==5){
//                      message_send_debugvalue(COMM_1, 0, mag.x);
//                      message_send_debugvalue(COMM_1, 1, mag.y);
//                      message_send_debugvalue(COMM_1, 2, mag.z);
//                      k=0;
```

```
//                }
//                k++;

                  #endif
        }
}


static inline void house_keeping(void){
        global_data.battery_voltage=battery_get_value();
        global_data.cpu_usage=(uint16_t)((sys_time_get_timer_counter()-
            sys_time_get_period_start_time())*10000/PERIODIC_TASK_PERIOD);
}

static inline void house_keeping_v2(void){
        global_data.battery_voltage=battery_get_value();
}


static inline void debug(void)
{
        uint8_t buffer[200];

//      sprintf(buffer, "%06d,%06d,%06d,\r\n", (int)(global_data.gyros_si.x
    *1000), (int)(global_data.gyros_si.y*1000), (int)(global_data.gyros_si.z
    *1000));
//      message_send_debug(COMM_1, buffer);

        sprintf(buffer, "%06d,%06d,%06d,%06d,%06d\r\n", (int) global_data.
            accel_raw.x,
                        (int) global_data.accel_raw.y, (int)global_data.
                            accel_raw.z, (int)global_data.supersonic_raw, (int)
                            global_data.pressure_raw);
        message_send_debug(COMM_1, buffer);
//
//      sprintf(buffer, "%06d,%06d,%06d\r\n", (int)(global_data.attitude.x
    *180/3.14),
//                      (int)(global_data.attitude.y*180/3.14), (int)(
    global_data.attitude.z*180/3.14));
//      message_send_debug(COMM_1, buffer);

//      sprintf(buffer, "%06d,%06d,%06d,", (int) global_data.gyros_raw.x, (int)
     global_data.gyros_raw.y, (int) global_data.gyros_raw.z);
//      message_send_debug(COMM_1, buffer);
//
//      sprintf(buffer, "%06d,%06d,%06d\r\n", (int) global_data.magnet_raw.x, (
    int) global_data.magnet_raw.y, (int) global_data.magnet_raw.z);
//      message_send_debug(COMM_1, buffer);

}


/**
 * @brief Check for periodic counter timeout
 *
 * This function can be called to check whether the counter associated with
     counter_id
```

```
 * has expired. If expired it the counter is set to the current time plus
     parameter ms
 * and TRUE is returned.
 */
static inline uint8_t ms_run_every(uint32_t ms, counter_id_t counter_id)
{
        uint32_t us = 1000*ms;
        uint32_t current_time = sys_time_clock_get_time_usec();
        if (next_exec_time[counter_id] <= current_time || next_exec_time[
            counter_id] - current_time > us) {
                next_exec_time[counter_id] = current_time+us;
                return 1;
        }
        else {
                return 0;
        }
}

/**
 * @brief Main function
 *
 * The main function first calls the main_init() function and then loops
     through
 * the main_periodic() function until shutdown.
 */
int main(void) {
        main_init();
        main_loop();
        return 0;
}
```

C.2. Reading the Sensors (sensors.h)

```
/*
 * sensors.h
 *
 *  Created on: Jun 15, 2009
 *      Author: mavteam, Christian Dobler
 */

#ifndef SENSORS_H_
#define SENSORS_H_

#include "conf.h"
//#include "max1168.h"
#include "sca3100.h"
#include "sca3000.h"
#include "bmp085.h"
#include "ads8341.h"
#include "ms2100.h"
#include "LPC21xx.h"
#include "global_data.h"
#include "spi.h"
#include "i2c.h"
#include "sys_time.h"
#include "inttypes.h"
#include "adc.h"
#include "dac.h"
#include <stdio.h>

#include "comm.h"
#include "led.h"
#include "generated/messages.h"


#define BURST_TIMES                 1

#ifdef BOARD_TWOG_BOOZ
static inline void sensors_init(void)
{
        spi_init();
        ms2100_init();
        max1168_init();
}

static inline void sensors_read_out(void)
{

        ms2100_send_request();
        while(ms2100_get_status()!=MS2100_IDLE);
        global_data.magnet_raw.x=ms2100_get_value(MS2100_X_AXIS);
        global_data.magnet_raw.y=ms2100_get_value(MS2100_Y_AXIS);
        global_data.magnet_raw.z=ms2100_get_value(MS2100_Z_AXIS);


        int temp;

        temp=0;
```

```
        for(int i=0; i<BURST_TIMES; i++){
                max1168_read_channel(GYROS_ROLL_MAX1168_CHANNEL);
                temp+=max1168_get_value(GYROS_ROLL_MAX1168_CHANNEL);
        }
        global_data.gyros_raw.x=temp/BURST_TIMES;

        temp=0;
        for(int i=0; i<BURST_TIMES; i++){
                max1168_read_channel(GYROS_PITCH_MAX1168_CHANNEL);
                temp+=max1168_get_value(GYROS_PITCH_MAX1168_CHANNEL);
        }
        global_data.gyros_raw.y=temp/BURST_TIMES;

        temp=0;
        for(int i=0; i<BURST_TIMES; i++){
                max1168_read_channel(GYROS_YAW_MAX1168_CHANNEL);
                temp+=max1168_get_value(GYROS_YAW_MAX1168_CHANNEL);
        }
        global_data.gyros_raw.z=temp/BURST_TIMES;

        temp=0;
        for(int i=0; i<BURST_TIMES; i++){
                max1168_read_channel(ACC_X_MAX1168_CHANNEL);
                temp+=max1168_get_value(ACC_X_MAX1168_CHANNEL);
        }
        global_data.accel_raw.x=temp/BURST_TIMES;

        temp=0;
        for(int i=0; i<BURST_TIMES; i++){
                max1168_read_channel(ACC_Y_MAX1168_CHANNEL);
                temp+=max1168_get_value(ACC_Y_MAX1168_CHANNEL);
        }
        global_data.accel_raw.y=temp/BURST_TIMES;

        temp=0;
        for(int i=0; i<BURST_TIMES; i++){
                max1168_read_channel(ACC_Z_MAX1168_CHANNEL);
                temp+=max1168_get_value(ACC_Z_MAX1168_CHANNEL);
        }
        global_data.accel_raw.z=temp/BURST_TIMES;

        global_data.supersonic_raw=adc_get_supersonic_value();

}
#endif

#if defined(IMU_PIXHAWK_V200) || defined(IMU_PIXHAWK_V210)

uint8_t mag_axis;
uint8_t bmp085_measurement_mode;

static inline void sensors_init(void)
{
        adc_init();
        spi_init();
        i2c_init();
        //ms2100_init();
```

```
        ads8341_init();
        sca3100_init();
        //bmp085_init();
        mag_axis = 0;
        bmp085_init();
        bmp085_measurement_mode = 255;
}

static inline void sensors_read_out_50hz(void)
{
        //ms2100 is read out parallel to the main loop (in each loop
        //one loop at a time)
        switch (mag_axis){
        case 0:
                global_data.magnet_raw.z=ms2100_get_value(MS2100_Z_AXIS);
                ms2100_send_request(MS2100_X_AXIS);
                mag_axis=1;
                break;
        case 1:
                global_data.magnet_raw.x=ms2100_get_value(MS2100_X_AXIS);
                ms2100_send_request(MS2100_Y_AXIS);
                mag_axis=2;
                break;
        case 2:
                global_data.magnet_raw.y=ms2100_get_value(MS2100_Y_AXIS);
                ms2100_send_request(MS2100_Z_AXIS);
                mag_axis=0;
                break;
        }


}

static inline void sensors_read_out_100hz(void)
{

        switch (bmp085_measurement_mode){
        case 255:
                bmp085_start_temp_measurement();
                bmp085_measurement_mode = 0;
                break;
        case 50:
                global_data.pressure_raw = bmp085_get_pressure();
                bmp085_start_temp_measurement();
                bmp085_measurement_mode = 0;
                break;
        default:
                if(bmp085_measurement_mode == 0) {
                        global_data.temperature = bmp085_get_temperature();
                }
                else{
                        global_data.pressure_raw = bmp085_get_pressure();
                }
                bmp085_start_pressure_measurement();
                bmp085_measurement_mode++;
        }
```

```
}


static inline void sensors_read_out_200hz(void)
{

        ads8341_read(ADS8341_0, GYROS_ROLL_ADS8341_0_CHANNEL);
        while (spi_running());
        ads8341_get_value(ADS8341_0,GYROS_ROLL_ADS8341_0_CHANNEL);
        global_data.gyros_raw.x = ads8341_get_value(ADS8341_0,
            GYROS_ROLL_ADS8341_0_CHANNEL);

        ads8341_read(ADS8341_0, GYROS_PITCH_ADS8341_0_CHANNEL);
        while (spi_running());
        global_data.gyros_raw.y = ads8341_get_value(ADS8341_0,
            GYROS_PITCH_ADS8341_0_CHANNEL);

        ads8341_read(ADS8341_0, GYROS_YAW_ADS8341_0_CHANNEL);
        while (spi_running());
        global_data.gyros_raw.z = ads8341_get_value(ADS8341_0,
            GYROS_YAW_ADS8341_0_CHANNEL);


        sca3100_read_res();
        while(spi_running());
        global_data.accel_raw.x = (int)sca3100_get_value(SCA3100_X_AXIS);
        global_data.accel_raw.y = (int)sca3100_get_value(SCA3100_Y_AXIS);
        global_data.accel_raw.z = (int)sca3100_get_value(SCA3100_Z_AXIS);

}

#endif


#ifdef BOARD_PIXHAWK_V100

static inline void pressure_init(void){
        uint16_t dac_value=200;
        uint16_t pressure_adc=0;
        while(pressure_adc<25000){
                dac_set(dac_value);
                sys_time_wait(100);
                ads8341_read(ADS8341_0, PRESSURE_ADS8341_0_CHANNEL);
                while (spi_running());
                pressure_adc = ads8341_get_value(ADS8341_0,
                    PRESSURE_ADS8341_0_CHANNEL);
                dac_value+=1;
        }
}

static inline void sensors_init(void)
{
        adc_init();
        dac_init();
        spi_init();
        ms2100_init();
        ads8341_init();
```

```
        sca3000_init();
        pressure_init();
}

static inline void sensors_read_out(void)
{
        //ms2100 is read out parallel to the main loop (in each loop
        //one loop at a time)
        static uint8_t mag_axis = 0;
        switch (mag_axis){
        case 0:
                global_data.magnet_raw.z=ms2100_get_value(MS2100_Z_AXIS);
                ms2100_send_request(MS2100_X_AXIS);
                mag_axis=1;
                break;
        case 1:
                global_data.magnet_raw.x=ms2100_get_value(MS2100_X_AXIS);
                ms2100_send_request(MS2100_Y_AXIS);
                mag_axis=2;
                break;
        case 2:
                global_data.magnet_raw.y=ms2100_get_value(MS2100_Y_AXIS);
                ms2100_send_request(MS2100_Z_AXIS);
                mag_axis=0;
                break;
        }

        ads8341_read(ADS8341_0, GYROS_ROLL_ADS8341_0_CHANNEL);
        while (spi_running());
        ads8341_get_value(ADS8341_0,GYROS_ROLL_ADS8341_0_CHANNEL);
        global_data.gyros_raw.x = ads8341_get_value(ADS8341_0,
            GYROS_ROLL_ADS8341_0_CHANNEL);

        ads8341_read(ADS8341_0, GYROS_PITCH_ADS8341_0_CHANNEL);
        while (spi_running());
        global_data.gyros_raw.y = ads8341_get_value(ADS8341_0,
            GYROS_PITCH_ADS8341_0_CHANNEL);

        ads8341_read(ADS8341_0, PRESSURE_ADS8341_0_CHANNEL);
        while (spi_running());
        global_data.pressure_raw =ads8341_get_value(ADS8341_0,
            PRESSURE_ADS8341_0_CHANNEL);

        ads8341_read(ADS8341_1, GYROS_YAW_ADS8341_1_CHANNEL);
        while (spi_running());
        ads8341_get_value(ADS8341_1,GYROS_YAW_ADS8341_1_CHANNEL);
        global_data.gyros_raw.z = ads8341_get_value(ADS8341_1,
            GYROS_YAW_ADS8341_1_CHANNEL);


        sca3000_read_res();
        while(spi_running());
        global_data.accel_raw.x = (int)sca3000_get_value(SCA3000_X_AXIS);
        global_data.accel_raw.y = (int)sca3000_get_value(SCA3000_Y_AXIS);
        global_data.accel_raw.z = (int)sca3000_get_value(SCA3000_Z_AXIS);
```

```
        global_data.supersonic_raw = adc_get_value(ADC_BPLANCD_IO_2_CHANNEL)
            *127/10;
}

#endif


#endif /* SENSORS_H_ */
```

C.3. I²C Subsystem

C.3.1. i2c.h

```
/*==========================================================================

PIXHAWK Micro Air Vehicle Flying Robotics Toolkit

(c) 2009, 2010 PIXHAWK PROJECT  <http://pixhawk.ethz.ch>

This file is part of the PIXHAWK project

    PIXHAWK is free software: you can redistribute it and/or modify
    it under the terms of the GNU General Public License as published by
    the Free Software Foundation, either version 3 of the License, or
    (at your option) any later version.

    PIXHAWK is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
    GNU General Public License for more details.

    You should have received a copy of the GNU General Public License
    along with PIXHAWK. If not, see <http://www.gnu.org/licenses/>.

==========================================================================*/

/**
 * @file i2c.h
 *   @brief I2C interface driver
 *
 *   This file contains the I2C driver.
 *   @author Christian Dobler <mavteam@student.ethz.ch>
 *
 */

#ifndef I2C_H_
#define I2C_H_

#include "inttypes.h"
#include "conf.h"

#if defined(IMU_PIXHAWK_V200) || defined(IMU_PIXHAWK_V210)

#define I2C_WRITE                        0              ///< definition of the
    R/W bit for a I2C write operation
#define I2C_READ                         1              ///< definition of the
    R/W bit for a I2C read operation

/**
 * @struct i2c_package This structure holds all the data necessary for a
     complete I2C operation.
 */
typedef struct {

        /*! pointer to data array*/
        uint8_t data[MAX_I2C_PACKAGE_SIZE];
```

```
        /*! the number of bytes in the data array */
        uint8_t length;

        /*! direction of I2C Operation: write: 0, read: 1 */
        uint8_t direction;

        /*! address of slave device to communicate with */
        uint8_t slave_address;

        /*! number of the I2C bus the addresses slave device is attached to: 0
            or 1 */
        uint8_t bus_number;

        /*! set this variable to 1, if a repeated start should be made between
            a write
         * package and the following read package */
        uint8_t write_read;

        /*! function to be defined by the specific device driver to handle the
            received
         *  data at the end of a I2C read operation */
        void (*i2c_done_handler)(void * package);

} i2c_package;


/**
 * @brief Initializes the I2C interfaces
 *
 * This function initializes the I2C subsystem. It has to be executed before
    any
 * I2C device driver.
 *
 */
void i2c_init(void);

/**
 * @brief Write and Read function for reading specific addresses in I2C devices
 *
 * This function first issues a I2C write package followed by an I2C read
    package
 * with a repeated start condition in between. Use this function to read a
    specific
 * address from an I2C device by first transmitting the read address in the
    write
 * package and then really reading the data with the read package.
 *
 * @param package_write         I2C write package (direction variable set to 0)
 * @param package_read          I2C read package (direction variable set to 1)
 */
void i2c_write_read(i2c_package * package_write, i2c_package * package_read);


/**
 * @brief Function to start an I2C operation
 *
```

```
 * This function should by called by I2C device drivers to start an I2C
    operation.
 * The details about the operation to be performed is passed to the I2C
    interface
 * driver in the I2C "package". See the definition of the structure i2c_package
 * for a detailed description.
 *
 * @param package       I2C package as defined in this file
 */
void i2c_op(i2c_package * package);

#endif

#endif
```

C.3.2. i2c.c

```
/*=========================================================================

PIXHAWK Micro Air Vehicle Flying Robotics Toolkit

(c) 2009, 2010 PIXHAWK PROJECT  <http://pixhawk.ethz.ch>

This file is part of the PIXHAWK project

    PIXHAWK is free software: you can redistribute it and/or modify
    it under the terms of the GNU General Public License as published by
    the Free Software Foundation, either version 3 of the License, or
    (at your option) any later version.

    PIXHAWK is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
    GNU General Public License for more details.

    You should have received a copy of the GNU General Public License
    along with PIXHAWK. If not, see <http://www.gnu.org/licenses/>.

=========================================================================*/

/**
 * @file i2c.h
 *   @brief I2C interface driver
 *
 *   This file contains the I2C driver.
 *   @author Christian Dobler <mavteam@student.ethz.ch>
 *
 */

#include "i2c.h"
#include "armVIC.h"
#include "LPC21xx.h"
#include "conf.h"

#include "comm.h"
```

```
#include "led.h"
#include "generated/messages.h"
#include "inttypes.h"
#include <stdio.h>
#include "uart.h"

#if defined(IMU_PIXHAWK_V200) || defined(IMU_PIXHAWK_V210)


static void I2C0_ISR(void) __attribute__((naked));                // interrupt
    handler for I2C Bus 0
static void I2C1_ISR(void) __attribute__((naked));                // interrupt
    handler for I2C Bus 1

/* function to start an I2C operation on I2C0 (not to be called by device
    drivers) */
void i2c0_op(i2c_package * package);
/* function to start an I2C operation on I2C1 (not to be called by device
    drivers) */
void i2c1_op(i2c_package * package);


i2c_package package_buffer0[I2C_PACKAGE_BUFFER_SIZE];    // package buffer for
    I2C0
i2c_package package_buffer1[I2C_PACKAGE_BUFFER_SIZE];    // package buffer for
    I2C1

uint8_t i2c_package_buffer0_insert_idx, i2c_package_buffer0_current_idx;
            // insertion and processing pointers for I2C0
uint8_t i2c_package_buffer1_insert_idx, i2c_package_buffer1_current_idx;
            // insertion and processing pointers for I2C1
uint8_t current_data_byte_i2c0, current_data_byte_i2c1;
                                // data byte counters
uint8_t i2c0_busy, i2c1_busy;
                                                    // I2C resource busy flags

uint16_t error_counter0, error_counter1;                                //
    counter variables to count number of errors per package


void i2c_init(void) {
        /* setup pins for I2C0 (SCL0, SDA0) and I2C1 (SCL1, SDA1) */
        PINSEL0 |= I2C0_PINSEL0_SCL  | I2C0_PINSEL0_SDA | I2C1_PINSEL0_SCL |
            I2C1_PINSEL0_SDA;

        /* setup I2C0 */
        I2C0ADR = LPC_I2C_ADR;
        I2C0SCLH = I2C0SCLH_VAL;
        I2C0SCLL = I2C0SCLL_VAL;
        I2C0CONSET = I2C0CONSET_VAL;

        /* initialize I2C0 interrupt vector */
        VICIntSelect &= ~VIC_BIT( VIC_I2C0 );                           /* I2C0
             selected as IRQ          */
        VICIntEnable = VIC_BIT( VIC_I2C0 );                             /*
            enable it                    */
        _VIC_CNTL(I2C_VIC_SLOT) = VIC_ENABLE | VIC_I2C0;
```

```
        _VIC_ADDR(I2C_VIC_SLOT) = (unsigned int)I2C0_ISR;    /* address of the
            I2C0 ISR  */

        /* setup I2C1 */
        I2C1ADR = LPC_I2C_ADR;
        I2C1SCLH = I2C1SCLH_VAL;
        I2C1SCLL = I2C1SCLL_VAL;
        I2C1CONSET = I2C1CONSET_VAL;

        /* initialize I2C1 interrupt vector */
        VICIntSelect &= ~VIC_BIT( VIC_I2C1 );                          /* I2C1
             selected as IRQ          */
        VICIntEnable = VIC_BIT( VIC_I2C1 );                            /*
            enable it                    */
        _VIC_CNTL(I2C_VIC_SLOT) = VIC_ENABLE | VIC_I2C1;
        _VIC_ADDR(I2C_VIC_SLOT) = (unsigned int)I2C1_ISR;    /* address of the
            I2C1 ISR  */

        // insertion and processing pointers for I2C0
        i2c_package_buffer0_insert_idx = 0;
        i2c_package_buffer0_current_idx = 0;
        i2c_package_buffer1_insert_idx = 0;
        i2c_package_buffer1_current_idx = 0;

        // status variables for I2C Bus 0 and 1
        i2c0_busy = 0;
        i2c1_busy = 0;

}

void i2c_write_read(i2c_package * package_write, i2c_package * package_read) {

        package_write->write_read = 1;
        package_read->write_read = 1;

        i2c_op(package_write);          // I2C wirite operation first
        i2c_op(package_read);           // just followed by I2C read operation

        return;
}

void i2c_op(i2c_package * package) {

        uint8_t buffer[200];     // string buffer for debug messages
        switch(package->bus_number) {           // decide wether device is on
            bus 0 or 1
                case 0:         i2c0_op(package);
                                        break;
                case 1:         i2c1_op(package);
                                        break;
                default:        sprintf((char *)buffer, "I2C error: non
                    existing bus number specified\n");
                                        message_send_debug(COMM_1, buffer);
        }

        return;
}
```

```
/* I2C0 bus operation starter function */
void i2c0_op(i2c_package * package) {

        // wait for free space, if buffer is full
        while(i2c_package_buffer0_current_idx-i2c_package_buffer0_insert_idx==1
             || (i2c_package_buffer0_current_idx==0 &&
            i2c_package_buffer0_insert_idx==(I2C_PACKAGE_BUFFER_SIZE-1))) ;

        package_buffer0[i2c_package_buffer0_insert_idx] = *package;

                    // add new i2c package to package buffer 0
    i2c_package_buffer0_insert_idx = (i2c_package_buffer0_insert_idx+1)%
        I2C_PACKAGE_BUFFER_SIZE;        // increment package insertion pointer

        if(!i2c0_busy) {                        // check whether I2C0 is busy:
            if busy, data will I2C operation will be executed later, but program
             execution can continue
                i2c0_busy = 1;                  // process locks I2C resource
                I2C0CONSET = 1<<STA;    // I2C start condition is generated on
                    the bus -> interrupt is generated when ready; next state
                    code: 0x08
                error_counter0 = 0;
        }

        return;
}

/* Interrupt handler for I2C0 interrupt */
static void I2C0_ISR(void) {

        ISR_ENTRY();                            // enter ISR

        uint8_t buffer[200];    // string buffer for debug messages
        if(error_counter0 > I2C_PERMANENT_ERROR_LIMIT){
                // TODO Lorenz: send error report to ground station
                sprintf((char *)buffer, "Permanent I2C error\n");
                message_send_debug(COMM_1, buffer);
        }

        switch(I2C0STAT) {                      // check current I2C0 state
        case 0x08:                                      // I2C start condition
            has been generated on the bus -> transmit slave address and read/
            write bit -> next state: 0x18 or 0x40
                I2C0DAT = package_buffer0[i2c_package_buffer0_current_idx].
                    slave_address | package_buffer0[
                    i2c_package_buffer0_current_idx].direction;
                I2C0CONCLR = 1<<STAC;
                I2C0CONCLR = 1<<SIC;    // clear I2C interrupt flag
                break;
        case 0x10:                                      // "repeated start"
            condition has been generated on the bus -> transmit slave address
            and read/write bit -> next state: 0x18 or 0x40
                I2C0DAT = package_buffer0[i2c_package_buffer0_current_idx].
                    slave_address | package_buffer0[
                    i2c_package_buffer0_current_idx].direction;
                I2C0CONCLR = 1<<STAC;
```

```
            I2C0CONCLR = 1<<SIC;
            break;
        case 0x18:                                      // slave address and
            write bit has been transmitted -> transmit first data byte -> next
            state: 0x28
            current_data_byte_i2c0 = 0;
            I2C0CONCLR = 1<<STAC;
            I2C0DAT = package_buffer0[i2c_package_buffer0_current_idx].data
                [current_data_byte_i2c0];
            current_data_byte_i2c0++;
            I2C0CONCLR = 1<<SIC;
            break;
        case 0x28:                                      // data byte has been
            transmitted
            // if there's another byte to be transmitted, do it
            if(current_data_byte_i2c0<package_buffer0[
                i2c_package_buffer0_current_idx].length) {
                    I2C0CONCLR = 1<<STAC;
                    I2C0DAT = package_buffer0[
                        i2c_package_buffer0_current_idx].data[
                        current_data_byte_i2c0];
                    current_data_byte_i2c0++;
                    I2C0CONCLR = 1<<SIC;
            }
            // it the last byte has been transmitted, check if there are
                unhandled packages in the package buffer
            else {
                    i2c_package_buffer0_current_idx = (
                        i2c_package_buffer0_current_idx+1)%
                        I2C_PACKAGE_BUFFER_SIZE; // increment pointer to
                        package in processing
                    error_counter0 = 0;
                    if(i2c_package_buffer0_current_idx==
                        i2c_package_buffer0_insert_idx) { // no unhandled
                        packages
                            I2C0CONCLR = 1<<STAC;
                            I2C0CONSET = 1<<STO;    // generate I2C stop
                                condition on the bus without restart
                            I2C0CONCLR = 1<<SIC;    // clear I2C interrupt
                                flag
                            i2c0_busy = 0;                  // release I2C
                                resource
                    }
                    else {                                          //
                        unhandled packages in package buffer
                            if(!(package_buffer0[
                                i2c_package_buffer1_current_idx].write_read
                                == 1)) {
                                    I2C0CONSET = 1<<STO;    // generate
                                        stop condition on the bus if no
                                        repeated start is necessary
                            }
                            I2C0CONSET = 1<<STA;    // generate "repeated
                                start" condition on the bus -> next state: 0
                                x10
                            I2C0CONCLR = 1<<SIC;    // clear I2C interrupt
                                flag
```

```
                }
            }
            break;
        case 0x20:                                      // I2C error state
            detection
            I2C0CONSET = 1<<STO;
            I2C0CONSET = 1<<STA;     // restart I2C state machine with
                current package
            error_counter0++;
            sprintf((char *)buffer, "I2C error: slave address not
                acknowledged (write)\n");
            message_send_debug(COMM1, buffer);
            I2C0CONCLR = 1<<SIC;
            break;
        case 0x30:                                      // I2C error state
            detection
            I2C0CONSET = 1<<STO;
            I2C0CONSET = 1<<STA;     // restart I2C state machine with
                current package
            error_counter0++;
            sprintf((char *)buffer, "I2C error: data not acknowledged\n");
            message_send_debug(COMM1, buffer);
            I2C0CONCLR = 1<<SIC;
            break;
        case 0x38:                                      // I2C error state
            detection
            I2C0CONSET = 1<<STA;     // restart I2C state machine with
                current package
            error_counter0++;
            sprintf((char *)buffer, "I2C error: arbitration lost\n");
            message_send_debug(COMM1, buffer);
            I2C0CONCLR = 1<<SIC;
            break;
        case 0x40:                                      // slave address and
            read bit has been transmitted -> clear interrupt and wait for first
            data byte -> next state: 0x50 or 0x58
            current_data_byte_i2c0 = 0;
            if (package_buffer0[i2c_package_buffer0_current_idx].length>1)
                I2C0CONSET = 1<<AA;       // if there's more than one byte to
                 be received -> next state: 0x50
            else I2C0CONCLR = 1<<AAC;

                                                // if there's only one byte
                to be received -> next state: 0x58
            I2C0CONCLR = 1<<SIC;     // clear I2C interrupt flag
            break;
        case 0x50:                                      // data byte has been
            received
            package_buffer0[i2c_package_buffer0_current_idx].data[
                current_data_byte_i2c0] = I2C0DAT;        // copy data byte
                to data array in I2C package
            current_data_byte_i2c0++;

                                                                // increment
                 data byte
```

```
            if((current_data_byte_i2c0+1)<package_buffer0[
                i2c_package_buffer0_current_idx].length) {        // there's
                 more than one byte left to be received
                    I2C0CONSET = 1<<AA;              // acknowledge next
                        data byte -> next state: 0x50
            }
            else {                                          // there's only
                 one byte left to be received
                    I2C0CONCLR = 1<<AAC;     // do not acknowledge next data
                         byte -> next state: 0x58
            }
            I2C0CONCLR = 1<<SIC;
            break;
        case 0x58:                                      // last data byte has
            been received

            package_buffer0[i2c_package_buffer0_current_idx].data[
                current_data_byte_i2c0] = I2C0DAT;        // copy data byte
                to data array in I2C package
            I2C0CONSET = 1<<STO;     // generate STOP condition on the I2C
                bus
            // execute I2C package processing function defined by driver
                for specific I2C devices
            if(package_buffer0[i2c_package_buffer0_current_idx].
                i2c_done_handler != NULL) {                       // check
                 if there is a package handler registered
                    package_buffer0[i2c_package_buffer0_current_idx].
                        i2c_done_handler(&(package_buffer0[
                        i2c_package_buffer0_current_idx])); // call package
                        handler
            }
            i2c_package_buffer0_current_idx = (
                i2c_package_buffer0_current_idx+1)%I2C_PACKAGE_BUFFER_SIZE;
                // increment pointer to package in processing
            error_counter0 = 0;
            // check if there are unhandled packages in the package buffer
            if(i2c_package_buffer0_current_idx==
                i2c_package_buffer0_insert_idx) {   // no unhandled packages
                    I2C0CONCLR = 1<<SIC;     // clear I2C interrupt flag
                    i2c0_busy = 0;                   // release I2C resource
            }
            else {                                          // unhandled
                packages in package buffer
                    I2C0CONSET = 1<<STA;     // generate "repeated start"
                        condition on the bus -> next state: 0x10
                    I2C0CONCLR = 1<<SIC;     // clear I2C interrupt flag
            }
            break;
        case 0x48:                                          // I2C error
            state detection
            I2C0CONSET = 1<<STO;
            I2C0CONSET = 1<<STA;             // restart I2C state machine
                with current package
            error_counter0++;
            sprintf((char *)buffer, "I2C error: slave address not
                acknowledged (read)\n");
            message_send_debug(COMM_1, buffer);
```

```
                I2C0CONCLR = 1<<SIC;
                break;

        default:                                                // I2C error
            state detection
                I2C0CONSET = 1<<STO;
                I2C0CONSET = 1<<STA;            // restart I2C state machine
                    with current package
                error_counter0++;
                sprintf((char *)buffer, "I2C error: undefined I2C state\n");
                message_send_debug(COMM_1, buffer);
                I2C0CONCLR = 1<<SIC;
        }

        VICVectAddr = 0x00000000;       // clear this interrupt from the VIC
        ISR_EXIT();                                     // exit ISR
}


/* the same for I2C1 */

/* I2C1 bus operation starter function */
void i2c1_op(i2c_package * package) {

        // wait for free space, if buffer is full
        while(i2c_package_buffer1_current_idx-i2c_package_buffer1_insert_idx==1
             || (i2c_package_buffer1_current_idx==0 &&
            i2c_package_buffer1_insert_idx==(I2C_PACKAGE_BUFFER_SIZE-1))) ;

        package_buffer1[i2c_package_buffer1_insert_idx] = *package;

                    // add new i2c package to package buffer 1
    i2c_package_buffer1_insert_idx = (i2c_package_buffer1_insert_idx+1)%
        I2C_PACKAGE_BUFFER_SIZE;        // increment package insertion pointer

        if(!i2c1_busy) {                        // check whether I2C1 is busy:
            if busy, data will I2C operation will be executed later, but program
             execution can continue
                i2c1_busy = 1;                  // process locks I2C resource
                I2C1CONSET = 1<<STA;    // I2C start condition is generated on
                    the bus -> interrupt is generated when ready; next state
                    code: 0x08
                error_counter1 = 0;
        }

        return;
}

/* Interrupt handler for I2C1 interrupt */
static void I2C1_ISR(void) {

        //sprintf((char*)buffer, "h ");
        //message_send_debug(COMM_1, buffer);
        uint8_t buffer[200];    // string buffer for debug messages
        ISR_ENTRY();

        if(error_counter1 > I2C_PERMANENT_ERROR_LIMIT){
```

```
        // TODO Lorenz: send error report to ground station
        sprintf((char *)buffer, "Permanent I2C error\n");
        message_send_debug(COMM_1, buffer);
}

switch(I2C1STAT) {                          // check current I2C1 state
case 0x08:                                          // I2C start condition
    has been generated on the bus -> transmit slave address and read/
    write bit -> next state: 0x18 or 0x40
        I2C1DAT = package_buffer1[i2c_package_buffer1_current_idx].
            slave_address | package_buffer1[
            i2c_package_buffer1_current_idx].direction;
        I2C1CONCLR = 1<<STAC;    // do not restart (continue normal I2C
            operation)
        I2C1CONCLR = 1<<SIC;     // clear I2C interrupt flag
        break;
case 0x10:                                          // "repeated start"
    condition has been generated on the bus -> transmit slave address
    and read/write bit -> next state: 0x18 or 0x40
        I2C1DAT = package_buffer1[i2c_package_buffer1_current_idx].
            slave_address | package_buffer1[
            i2c_package_buffer1_current_idx].direction;
        I2C1CONCLR = 1<<STAC;
        I2C1CONCLR = 1<<SIC;
        break;
case 0x18:                                          // slave address and
    write bit has been transmitted -> transmit first data byte -> next
    state: 0x28
        current_data_byte_i2c1 = 0;
        I2C1CONCLR = 1<<STAC;
        I2C1DAT = package_buffer1[i2c_package_buffer1_current_idx].data
            [current_data_byte_i2c1];
        current_data_byte_i2c1++;
        I2C1CONCLR = 1<<SIC;
        break;
case 0x28:                                          // data byte has been
    transmitted
        // if there's another byte to be transmitted, do it
        if(current_data_byte_i2c1<package_buffer1[
            i2c_package_buffer1_current_idx].length) {
                I2C1CONCLR = 1<<STAC;
                I2C1DAT = package_buffer1[
                    i2c_package_buffer1_current_idx].data[
                    current_data_byte_i2c1];
                current_data_byte_i2c1++;
                I2C1CONCLR = 1<<SIC;
        }
        // it the last byte has been transmitted, check if there are
            unhandled packages in the package buffer
        else {
                i2c_package_buffer1_current_idx = (
                    i2c_package_buffer1_current_idx+1)%
                    I2C_PACKAGE_BUFFER_SIZE;  // increment pointer to
                    package in processing
                error_counter1 = 0;
                if(i2c_package_buffer1_current_idx==
                    i2c_package_buffer1_insert_idx) {   // no unhandled
```

```
                    packages
                        I2C1CONCLR = 1<<STAC;
                        I2C1CONSET = 1<<STO;     // generate I2C stop
                            condition on the bus without restart
                        I2C1CONCLR = 1<<SIC;     // clear I2C interrupt
                            flag
                        i2c1_busy = 0;                  // release I2C
                            resource
                }
                else {                                          //
                    unhandled packages in package buffer
                        if(!(package_buffer1[
                            i2c_package_buffer1_current_idx].write_read
                            == 1)) {
                                I2C1CONSET = 1<<STO;     // generate
                                    stop condition on the bus if no
                                    repeated start is necessary
                        }
                        I2C1CONSET = 1<<STA;            // generate "
                            repeated start" condition on the bus -> next
                             state: 0x10
                        I2C1CONCLR = 1<<SIC;            // clear I2C
                            interrupt flag
                }
            }
            break;
    case 0x20:
                // I2C error state detection
            I2C1CONSET = 1<<STO;
            I2C1CONSET = 1<<STA;                        // restart I2C
                state machine with current package
            error_counter1++;
            sprintf((char *)buffer, "I2C error: slave address not
                acknowledged (write)\n");
            message_send_debug(COMM1, buffer);
            I2C1CONCLR = 1<<SIC;
            break;
    case 0x30:
                // I2C error state detection
            I2C1CONSET = 1<<STO;
            I2C1CONSET = 1<<STA;                        // restart I2C
                state machine with current package
            error_counter1++;
            sprintf((char *)buffer, "I2C error: data not acknowledged\n");
            message_send_debug(COMM1, buffer);
            I2C1CONCLR = 1<<SIC;
            break;
    case 0x38:
                // I2C error state detection
            I2C1CONSET = 1<<STA;                        // restart I2C
                state machine with current package
            error_counter1++;
            sprintf((char *)buffer, "I2C error: arbitration lost\n");
            message_send_debug(COMM1, buffer);
            I2C1CONCLR = 1<<SIC;
            break;
```

```
case 0x40:
            // slave address and read bit has been transmitted -> clear
    interrupt and wait for first data byte -> next state: 0x50 or 0x58
        current_data_byte_i2c1 = 0;
        if (package_buffer1[i2c_package_buffer1_current_idx].length>1)
            I2C1CONSET = 1<<AA;       // if there's more than one byte to
             be received -> next state: 0x50
        else I2C1CONCLR = 1<<AAC;

                                          // if there's only one byte
            to be received -> next state: 0x58
        I2C1CONCLR = 1<<SIC;                              // clear I2C
            interrupt flag
        break;
case 0x50:
            // data byte has been received
        package_buffer1[i2c_package_buffer1_current_idx].data[
            current_data_byte_i2c1] = I2C1DAT;         // copy data byte
            to data array in I2C package
        current_data_byte_i2c1++;

                                                          // increment
             data byte
        if((current_data_byte_i2c1+1)<package_buffer1[
            i2c_package_buffer1_current_idx].length) {         // there's
             more than one byte left to be received
                I2C1CONSET = 1<<AA;                                //
                    acknowledge next data byte -> next state: 0x50
        }
        else {
                    // there's only one byte left to be received
                I2C1CONCLR = 1<<AAC;                      // do not
                    acknowledge next data byte -> next state: 0x58
        }
        I2C1CONCLR = 1<<SIC;
        break;
case 0x58:
            // last data byte has been received
        package_buffer1[i2c_package_buffer1_current_idx].data[
            current_data_byte_i2c1] = I2C1DAT;         // copy data byte
            to data array in I2C package
        I2C1CONSET = 1<<STO;                              // generate
            STOP condition on the I2C bus
        //uart0_transmit(package_buffer1[
            i2c_package_buffer1_current_idx].data[current_data_byte_i2c1
            ]);
        if(package_buffer1[i2c_package_buffer1_current_idx].
            i2c_done_handler != NULL) {          // check if there is a
            package handler registered
                package_buffer1[i2c_package_buffer1_current_idx].
                    i2c_done_handler(&(package_buffer1[
                    i2c_package_buffer1_current_idx])); // call package
                    handler
        }
        i2c_package_buffer1_current_idx = (
            i2c_package_buffer1_current_idx+1)%I2C_PACKAGE_BUFFER_SIZE;
              // increment pointer to package in processing
```

```
                error_counter1 = 0;
                // check if there are unhandled packages in the package buffer
                if(i2c_package_buffer1_current_idx==
                    i2c_package_buffer1_insert_idx) {   // no unhandled packages
                        I2C1CONCLR = 1<<SIC;     // clear I2C interrupt flag
                        i2c1_busy = 0;                   // release I2C resource
                }
                else {                                          // unhandled
                    packages in package buffer
                        I2C1CONSET = 1<<STA;     // generate "repeated start"
                            condition on the bus -> next state: 0x10
                        I2C1CONCLR = 1<<SIC;     // clear I2C interrupt flag
                }
                break;
        case 0x48:                                              // I2C error
            state detection
                I2C1CONSET = 1<<STO;
                I2C1CONSET = 1<<STA;             // restart I2C state machine
                    with current package
                error_counter1++;
                sprintf((char *)buffer, "I2C error: slave address not
                    acknowledged (read)\n");
                message_send_debug(COMM_1, buffer);
                I2C1CONCLR = 1<<SIC;
                break;

        default:                                                // I2C error
            state detection
                I2C1CONSET = 1<<STO;
                I2C1CONSET = 1<<STA;             // restart I2C state machine
                    with current package
                error_counter1++;
                sprintf((char *)buffer, "I2C error: undefined I2C state\n");
                message_send_debug(COMM_1, buffer);
                I2C1CONCLR = 1<<SIC;
        }

        VICVectAddr = 0x00000000;        // clear this interrupt from the VIC
        ISR_EXIT();                                     // exit ISR
}

#endif
```

C.4. BMP085 Pressure Sensor Device Driver

C.4.1. bmp085.h

```
/*=========================================================================
PIXHAWK Micro Air Vehicle Flying Robotics Toolkit

(c) 2009, 2010 PIXHAWK PROJECT  <http://pixhawk.ethz.ch>

This file is part of the PIXHAWK project

    PIXHAWK is free software: you can redistribute it and/or modify
    it under the terms of the GNU General Public License as published by
    the Free Software Foundation, either version 3 of the License, or
    (at your option) any later version.

    PIXHAWK is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
    GNU General Public License for more details.

    You should have received a copy of the GNU General Public License
    along with PIXHAWK. If not, see <http://www.gnu.org/licenses/>.

=========================================================================*/

/*
 * @file
 *   @brief BMP085 pressure sensor device driver
 *
 *   This file contains the device driver for the Bosch BMP085 pressure sensor.
 *   @author Christian Dobler <mavteam@student.ethz.ch>
 *
 */

#ifndef BMP085_H_
#define BMP085_H_

#include "inttypes.h"
#include "i2c.h"

#if defined(IMU_PIXHAWK_V200) || defined(IMU_PIXHAWK_V210)

/**
 * @struct bmp085_smd500_calibration_param_t This structure holds all device
     specific calibration parameters.
 */
typedef struct {
   short ac1;
   short ac2;
   short ac3;
   unsigned short ac4;
   unsigned short ac5;
   unsigned short ac6;
   short b1;
   short b2;
```

```
    short mb;
    short mc;
    short md;
} bmp085_smd500_calibration_param_t;


/**
 * @struct bmp085_t This struct holds all control, measurement and status data
     of the BMP085 pressure sensor.
 */
typedef struct {
        bmp085_smd500_calibration_param_t cal_param;    ///< structure that
            holds calibration data
        uint8_t chip_id, ml_version, al_version;                ///< chip
            identification data
        uint8_t dev_addr;
                            ///< I2C slave address of BMP085
        uint8_t bus_number;
                            ///< number of the I2C bus the BMP085 is connected
            to
        uint8_t sensortype;
                            ///< value of sensor type (should be 0x55)

        int32_t param_b5;
                            ///< calibration parameter calculated by
            bmp085_get_temperature()
        uint8_t oversampling_setting;                                   ///<
            setting of the pressure conversion mode (internal oversampling)

        uint32_t current_pressure;
                    ///< value of the most recent pressure measurement
        uint32_t current_temp;
                    ///< value of the most recent temperature measurement

        uint8_t busy;
                            ///< set to 1 during measurements
        uint8_t measurement_type;
                    ///< temporarily stores current measurement type (for driver
             internal use only)
} bmp085_t;


/*  Settings for pressure sensor */

/** BMP085 I2C slave addresses */
#define BMP085_I2C_ADDR                         0xEE            ///< I2C slave
    address of the BMP085

/** BMP085 register addresses */
#define BMP085_CONTROL_REG                      0xF4            ///< address of
     the BMP085 control register
#define BMP085_DATA_MSB_REG             0xF6            ///< address of the
    BMP085 data (MSB) register
#define BMP085_DATA_LSB_REG             0xF7            ///< address of the
    BMP085 data (LSB) register
#define BMP085_DATA_XLSB_REG            0xF7            ///< address of the
    BMP085 data (XLSB) register (optional for ultra high resolution mode)
```

```
/** BMP085 control register settings */
#define BMP085_T_MEASURE                0x2E            ///< writing this value
    to the BMP085 control register starts temperature measurement
#define BMP085_P_MEASURE                0x34            ///< writing this value
    to the BMP085 control register starts pressure measurement


#define BMP085_CHIP_ID                          0x55            ///< ID value
    of every BMP085 chip
#define BOSCH_PRESSURE_BMP085           85                      ///< ID value
    in decimal representation

#define BMP085_PROM_START__ADDR         0xAA            ///< start address of
    the internal EEPROM containing the factory calibration data
#define BMP085_PROM_DATA__LEN           22                      ///< size of
    the internal EEPROM

#define BMP085_CHIP_ID_REG                      0xD0            ///< address of
    the register containing the chip ID
#define BMP085_VERSION_REG                      0xD1            ///< address of
    the register containing the chip version

/** defines to extract chip ID correctly */
#define BMP085_CHIP_ID__POS                     0
#define BMP085_CHIP_ID__MSK                     0xFF
#define BMP085_CHIP_ID__LEN                     8
#define BMP085_CHIP_ID__REG                     BMP085_CHIP_ID_REG

/** defines to extract ML version correctly */
#define BMP085_ML_VERSION__POS          0
#define BMP085_ML_VERSION__LEN          4
#define BMP085_ML_VERSION__MSK          0x0F
#define BMP085_ML_VERSION__REG          BMP085_VERSION_REG

/** defines to extract AL version correctly */
#define BMP085_AL_VERSION__POS          4
#define BMP085_AL_VERSION__LEN          4
#define BMP085_AL_VERSION__MSK          0xF0
#define BMP085_AL_VERSION__REG          BMP085_VERSION_REG

/** bmp085.sensortype value if no device detected */
#define E_SENSOR_NOT_DETECTED           0

/** standard calibration parameters */
#define BMP085_PARAM_MG                 3038        ///<calibration parameter
#define BMP085_PARAM_MH                 -7357       ///<calibration parameter
#define BMP085_PARAM_MI                 3791        ///<calibration parameter


/** conversion timeouts for different pressure modes */
#define BMP085_PRESS_0_CONVERSION_TIME          6000
#define BMP085_PRESS_1_CONVERSION_TIME          9000
#define BMP085_PRESS_2_CONVERSION_TIME          15000

/** conversion timeout for temperature measurement mode */
#define BMP085_TEMP_CONVERSION_TIME                     6000
```

```
/** control bit extraction function */
#define BMP085_GET_BITSLICE(regvar, bitname)\
                        (regvar & bitname##__MSK) >> bitname##__POS

#define BMP085_SET_BITSLICE(regvar, bitname, val)\
                    (regvar & ~bitname##__MSK) | ((val<<bitname##__POS)&bitname##
                        __MSK)


/**
 * @brief Initialization of the BMP085 pressure sensor
 *
 * This function initializes the BMP085 pressure sensor. The I2C subsystem
 * has to be initialized beforehand; otherwise this will fail.
 */
void bmp085_init(void);

/**
 * @brief Function to read the BMP085 chip ID register
 *
 * This function is invoked by the I2C subsystem during the initialization
 * of the BMP085 pressure sensor. This function does not need to be called
 * by application software.
 *
 * @param package I2C package as described in i2c.h
 */
static void bmp085_read_chip_id_on_init(i2c_package *package);

/**
 * @brief Function to read the BMP085 chip ID register
 *
 * This function is invoked by the I2C subsystem during the initialization
 * of the BMP085 pressure sensor. This function does not need to be called
 * by application software.
 *
 * @param package I2C package as described in i2c.h
 */
static void bmp085_read_version_on_init(i2c_package *package);

/**
 * @brief Starts read-out of factory calibration data from the sensor on init
 *
 * This function is invoked by the I2C subsystem during the initialization
 * of the BMP085 pressure sensor. This function does not need to be called
 * by application software.
 */
static void bmp085_get_cal_param(void);

/**
 * @brief Stores factory calibration data read out from the sensor on init
 *
 * This function is invoked by the I2C subsystem during the initialization
 * of the BMP085 pressure sensor. This function does not need to be called
 * by application software.
```

```
 *
 * @param package I2C package as described in i2c.h
 */
static void bmp085_store_cal_param_on_init(i2c_package *package);

/**
 * @brief Function to start temperature measurement
 *
 * This function has to be called by the application software to start a
     temperature
 * measurement on the BMP085. The BMP085 needs 4.5ms until the temperature
     measurement
 * is done. Measurement read-out is done by bmp085_start_measurement_read()
     called by
 * the EOC interrupt.
 *
 * This function needs to be called once before starting the first pressure
     measurement.
 */
void bmp085_start_temp_measurement(void);

/**
 * @brief Function to start temperature measurement
 *
 * This function has to be called by the application software to start a
     pressure
 * measurement on the BMP085. The BMP085 needs 4.5ms - 13.5ms (depending on the
     oversampling
 * setting selected by BMP085_PRESSURE_CONVERSION_MODE) until the temperature
     measurement is
 * done. Measurement read-out is done by bmp085_start_measurement_read() called
     by the EOC
 * interrupt.
 *
 * Execute one complete temperature measurement before calling this function.
 */
void bmp085_start_pressure_measurement(void);

/**
 * @brief Function to read out measurement result
 *
 * This function reads out the the 16 bit measurement result register from the
     BMP085.
 * It is invoked by the EOC interrupt (PH v200: timer interrupt, PH v210: EINT2
     ). The
 * application software should not call this function.
 */
void bmp085_start_measurement_read(void);

/**
 * @brief Stores measurement data read out from the sensor in the driver
 *
 * This function is invoked by the I2C subsystem during the initialization
 * of the BMP085 pressure sensor. This function does not need to be called
 * by application software.
 *
 * @param package I2C package as described in i2c.h
```

```
 */
static void bmp085_save_measurement(i2c_package *package);

/**
 * @brief Function to get the most recent temperature value
 *
 * This function has to be called by the application software to get the most
    recent
 * temperature value. This function returns immediately (no delays). Call this
    function
 * at least once before calling bmp085_get_pressure().
 *
 * @return 16-bit temperature value in 0.1°C
 */
int16_t bmp085_get_temperature(void);

/**
 * @brief Function to get the most recent pressure value
 *
 * This function has to be called by the application software to get the most
    recent
 * pressure value. This function returns immediately (no delays). Execute one
    complete
 * temperature measurement before calling this function.
 *
 * @return 32-bit pressure value in 1 Pa
 */
int32_t bmp085_get_pressure(void);
#endif // IMU_V200
#endif // BMP085_H_
```

C.4.2. bmp085.c

```
/*==========================================================================

PIXHAWK Micro Air Vehicle Flying Robotics Toolkit

(c) 2009, 2010 PIXHAWK PROJECT  <http://pixhawk.ethz.ch>

This file is part of the PIXHAWK project

    PIXHAWK is free software: you can redistribute it and/or modify
    it under the terms of the GNU General Public License as published by
    the Free Software Foundation, either version 3 of the License, or
    (at your option) any later version.

    PIXHAWK is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
    GNU General Public License for more details.

    You should have received a copy of the GNU General Public License
    along with PIXHAWK. If not, see <http://www.gnu.org/licenses/>.

==========================================================================*/
```

```
/**
 * @file
 *   @brief BMP085 pressure sensor device driver
 *
 *   This file contains the device driver for the Bosch BMP085 pressure sensor.
 *   @author Christian Dobler <mavteam@student.ethz.ch>
 *
 */

#include "inttypes.h"
#include "bmp085.h"
#include "conf.h"
#include "i2c.h"
#include "armVIC.h"
#include "LPC21xx.h"
#include "sys_time.h"

#include "comm.h"
#include "led.h"
#include "generated/messages.h"
#include <stdio.h>

#if defined(IMU_PIXHAWK_V200) || defined(IMU_PIXHAWK_V210)

#ifdef IMU_PIXHAWK_V210
static void EOC_EXTINT_ISR(void) __attribute__((naked));
#endif

bmp085_t bmp085;

void bmp085_init(void)
{
        i2c_package package_write, package_read;

        // set up static data in device descriptor structure bmp085
        bmp085.sensortype = E_SENSOR_NOT_DETECTED;                    // no
            sensor detected yet
        bmp085.dev_addr = BMP085_I2C_ADDR;                    // preset BMP085
            slave I2C_addr
        bmp085.bus_number = BMP085_I2C_BUS_NUMBER;                    //
            number of the I2C bus, that the BMP085 is connected to
        bmp085.oversampling_setting = BMP085_PRESSURE_CONVERSION_MODE;  // mode
             for sensor internal oversampling of pressure values
        bmp085.busy = 0;
                                    // reset BMP085 state variable
        bmp085.measurement_type = 0;
                    // no measurement type specified yet (1: temperature, 2:
            pressure)

        #ifdef IMU_PIXHAWK_V210
        /* configure EOC pin */
        /* connect BMP085_EOC pin to EINT2 */
        BMP085_EOC_PINSEL |= BMP085_EOC_PINSEL_VAL << BMP085_EOC_PINSEL_BIT;
        EXTMODE|= 1 << BMP085_EOC_EINT;                    /* EINT2 is edge
            triggered */
```

```
        EXTPOLAR|= 1 << BMP085_EOC_EINT;                    /* EINT2 is triggered
            on rising edge */
        EXTINT|=  1 << BMP085_EOC_EINT;                     /* clear pending EINT2
            */

        /* initialize interrupt vector */
        VICIntSelect &= ~VIC_BIT(VIC_EINT2);     /* select EINT2 as IRQ source
            */
        VICIntEnable = VIC_BIT(VIC_EINT2);                  /* enable it
                            */
        _VIC_CNTL(PRESSURE_EOC_VIC_SLOT) = VIC_ENABLE | VIC_EINT2;
        _VIC_ADDR(PRESSURE_EOC_VIC_SLOT) = (unsigned int) EOC_EXTINT_ISR; //
            address of the ISR
        #endif

        // first read chip ID register to certify that everything is correctly
            connected
        // set up I2C write package
        package_write.data[0] = BMP085_CHIP_ID__REG;     // address of BMP085
            chip ID register
        package_write.length = 1;
                    // just write the 8-bit address to be read
        package_write.direction = I2C_WRITE;
        package_write.slave_address = bmp085.dev_addr;  // I2C slave address of
             BM085
        package_write.bus_number = bmp085.bus_number;   // number of the I2C
            bus, that the BMP085 is connected to
        package_write.write_read = 1;                                   // make
             repeated start after this package to receive data
        package_write.i2c_done_handler = NULL;                  // nothing to
            be done at end of I2C write op

        // set up I2C read package
        package_read.length = 1;
        package_read.direction = I2C_READ;
        package_read.slave_address = bmp085.dev_addr;
        package_read.bus_number = bmp085.bus_number;
        package_read.write_read = 1;
        package_read.i2c_done_handler = &bmp085_read_chip_id_on_init; //
            bmp085_read_chip_id_on_init() is invoked at the end of the I2C
            operation

        i2c_write_read(&package_write, &package_read);  /* read Chip Id */


        return;
}

static void bmp085_read_chip_id_on_init(i2c_package *package) {

        i2c_package package_write, package_read;

        // extract chip ID from received I2C package
        bmp085.chip_id = BMP085_GET_BITSLICE(package->data[0], BMP085_CHIP_ID);
                    /* get bitslice */
```

```
        if (bmp085.chip_id == BMP085_CHIP_ID) {                         // that
            's what the chip ID should be

                bmp085.sensortype = BOSCH_PRESSURE_BMP085;              // set
                    sensor type to Bosch BMP085

                // now read out chip version
                // set up I2C write package
                package_write.data[0] = BMP085_VERSION_REG;             //
                    address of BMP085 version register
                package_write.length = 1;
                                    // just write the 8-bit address to be read
                package_write.direction = I2C_WRITE;
                package_write.slave_address = bmp085.dev_addr;
                package_write.bus_number = bmp085.bus_number;
                package_write.write_read = 1;
                            // make repeated start after this package to receive
                     data
                package_write.i2c_done_handler = NULL;

                // set up I2C read package
                package_read.length = 1;
                package_read.direction = I2C_READ;
                package_read.slave_address = bmp085.dev_addr;
                package_read.bus_number = bmp085.bus_number;
                package_read.write_read = 1;
                package_read.i2c_done_handler = &bmp085_read_version_on_init;
                       // bmp085_read_version_on_init() is invoked at the end of
                     the I2C operation

                i2c_write_read(&package_write, &package_read);  /* read Version
                     reg */

        }

        else {          // this should not happen, if everything is correctly
            connected
                uint8_t buffer[100];    // string buffer for debug messages
                sprintf((char *)buffer, "BMP085 error: wrong chip ID on bus\n")
                    ;
                message_send_debug(COMM_1, buffer);

        }

}

static void bmp085_read_version_on_init(i2c_package *package) {

        // extract version numbers from received I2C package
        bmp085.ml_version = BMP085_GET_BITSLICE(package->data[0],
            BMP085_ML_VERSION);         /* get ML Version */
        bmp085.al_version = BMP085_GET_BITSLICE(package->data[0],
            BMP085_AL_VERSION);             /* get AL Version */

        bmp085_get_cal_param( );                /* readout bmp085 calibration
            parameters */
```

```
}

static void bmp085_get_cal_param(void)
{
        i2c_package package_write, package_read;

        // initiate now readout of bmp085 calibration parameters

        package_write.data[0] = BMP085_PROM_START_ADDR;         // start
            address of the internal EEPROM of the BMP085
        package_write.length = 1;
                            // just write the 8-bit address to be read
        package_write.direction = I2C_WRITE;
        package_write.slave_address = bmp085.dev_addr;
        package_write.bus_number = bmp085.bus_number;
        package_write.write_read = 1;
        package_write.i2c_done_handler = NULL;

        package_read.length = BMP085_PROM_DATA_LEN;             // number of
            calibration bytes to be read fromthe BMP085
        package_read.direction = I2C_READ;
        package_read.slave_address = bmp085.dev_addr;
        package_read.bus_number = bmp085.bus_number;
        package_read.write_read = 1;
                    // make repeated start after this package to receive data
        package_read.i2c_done_handler = &bmp085_store_cal_param_on_init;
                    // bmp085_store_cal_param_on_init() is invoked at the end of
             the I2C operation
        i2c_write_read(&package_write, &package_read);  /* read calibration
            parameters over I2C */

}

// stores calibration data read out from the EEPROM in the sensor
static void bmp085_store_cal_param_on_init(i2c_package *package) {

        /*parameters AC1-AC6*/
        bmp085.cal_param.ac1 =  (package->data[0] <<8) | package->data[1];
        bmp085.cal_param.ac2 =  (package->data[2] <<8) | package->data[3];
        bmp085.cal_param.ac3 =  (package->data[4] <<8) | package->data[5];
        bmp085.cal_param.ac4 =  (package->data[6] <<8) | package->data[7];
        bmp085.cal_param.ac5 =  (package->data[8] <<8) | package->data[9];
        bmp085.cal_param.ac6 =  (package->data[10] <<8) | package->data[11];

        /*parameters B1,B2*/
        bmp085.cal_param.b1 =  (package->data[12] <<8) | package->data[13];
        bmp085.cal_param.b2 =  (package->data[14] <<8) | package->data[15];

        /*parameters MB,MC,MD*/
        bmp085.cal_param.mb =  (package->data[16] <<8) | package->data[17];
        bmp085.cal_param.mc =  (package->data[18] <<8) | package->data[19];
        bmp085.cal_param.md =  (package->data[20] <<8) | package->data[21];

        // init done

}
```

```
// start temperature measurement
void bmp085_start_temp_measurement()
{
        i2c_package package;

        // set up command package
        package.data[0] = BMP085_CONTROL_REG;    // write to the BMP085 control
            register
        package.data[1] = BMP085_T_MEASURE;              // the command to start
             a temperature measurement
        package.length = 2;                                              // that
             is 2 bytes
        package.direction = I2C_WRITE;
        package.slave_address = bmp085.dev_addr;
        package.bus_number = bmp085.bus_number;
        package.write_read = 0;                                  // make stop
            condition on I2C bus after this command
        package.i2c_done_handler = NULL;

        while(bmp085.busy) ;                                     // wait if
            BMP085 is already locked by other process

        bmp085.busy = 1;                                                 // lock
             BMP085 resource
        bmp085.measurement_type = 1;                     // set measurement type
             to 1 := temperature measurement
        i2c_op(&package);  /* write register to be read */

        // on a v2.1 IMU the BMP085_EOC interrupt will designate end-of-
            conversion of sensor value (= data is ready)

        #ifdef IMU_PIXHAWK_V200          // on a v2.0 IMU timer interrupt is
            used to specify end-of-conversion
        T0MCR |= TMCR_MR0_I;
        T0MR0 = T0TC + SYS_TICS_OF_USEC(BMP085_TEMP_CONVERSION_TIME);    //
            timer interrupt will be generated after temperature conversion
            timeout
        #endif

}

// start pressure measurement
void bmp085_start_pressure_measurement()
{
        i2c_package package;

        // set up command package
        package.data[0] = BMP085_CONTROL_REG + (bmp085.oversampling_setting <<
            6);       // write to the BMP085 control register
        package.data[1] = BMP085_P_MEASURE;                      // the command
            to start a pressure measurement
        package.length = 2;
                    // that is 2 bytes
        package.direction = I2C_WRITE;
        package.slave_address = bmp085.dev_addr;
        package.bus_number = bmp085.bus_number;
```

```
        package.write_read = 0;                                         // make
            stop condition on I2C bus after this command
        package.i2c_done_handler = NULL;

        while(bmp085.busy) ;                                            // wait
            if BMP085 is already locked by other process

        bmp085.busy = 1;
                    // lock BMP085 resource
        bmp085.measurement_type = 2;                            // set
            measurement type to 1 := temperature measurement

        i2c_op(&package);  /* write register to be read */

        // on a v2.1 IMU the BMP085_EOC interrupt will designate end-of-
            conversion of sensor value (= data is ready)

        #ifdef IMU_PIXHAWK_V200                         // on a v2.0 IMU timer
            interrupt is used to specify end-of-conversion
        T0MCR |= TMCR_MR0_I;                            // timer interrupt will
            be generated after temperature conversion timeout which depends on
            oversampling setting
        uint8_t buffer[100];    // string buffer for debug messages
        switch(bmp085.oversampling_setting) {
                case 0:
                        T0MR0 = T0TC + SYS_TICS_OF_USEC(
                            BMP085_PRESS_0_CONVERSION_TIME);
                        break;
                case 1:
                        T0MR0 = T0TC + SYS_TICS_OF_USEC(
                            BMP085_PRESS_1_CONVERSION_TIME);
                        break;
                case 2:
                        T0MR0 = T0TC + SYS_TICS_OF_USEC(
                            BMP085_PRESS_2_CONVERSION_TIME);
                        break;
                default:
                        sprintf((char *)buffer, "BMP085 error: non-existing
                            conversion mode defined\n");
                        message_send_debug(COMM_1, buffer);
        }
        #endif
}

// start reading out measurement from the sensor (invoked by either timer or
    EOC interrupt)
void bmp085_start_measurement_read() {

        i2c_package package_write, package_read;

        // set up I2C write package
        package_write.data[0] = BMP085_DATA_MSB_REG;    // register address to
            read the measurement
        package_write.length = 1;
                    // this command is one byte
        package_write.direction = I2C_WRITE;
        package_write.slave_address = bmp085.dev_addr;
```

```
        package_write.bus_number = bmp085.bus_number;
        package_write.write_read = 1;
        package_write.i2c_done_handler = NULL;

        // set up I2C read package
        package_read.length = 2;
                    // measurement is a 16 bit value = 2 bytes
        package_read.direction = I2C_READ;
        package_read.slave_address = bmp085.dev_addr;
        package_read.bus_number = bmp085.bus_number;
        package_read.write_read = 1;                                    // make
            repeated start after this package to receive data
        package_read.i2c_done_handler = &bmp085_save_measurement;       //
            bmp085_save_measurement() is invoked at the end of the I2C operation

        i2c_write_read(&package_write, &package_read);  // start I2C operation

}

// stores measurements temporarily within the driver
static void bmp085_save_measurement(i2c_package *package)
{
        uint8_t buffer[100];    // string buffer for debug messages
        switch(bmp085.measurement_type) {
                case 1:         // if temperature has been measured, store
                    temperature
                        bmp085.current_temp = (package->data[0] <<8) | package
                            ->data[1];
                        break;
                case 2:         // if pressure has been measured, store
                    temperature
                        bmp085.current_pressure = (package->data[0] <<8) |
                            package->data[1];
                        break;
                default:        // no other measurement modes available
                        sprintf((char *)buffer, "BMP085 error: no measurement
                            type defined\n");
                        message_send_debug(COMM_1, buffer);
        }

        bmp085.busy = 0;        // release locked BMP085 resource

}

// function to convert and scale temperature values to 0.1ï¿½C (taken from the
    BMP085 API from BOSCH)
int16_t bmp085_get_temperature(void)
{
        int16_t temperature;
        int32_t x1,x2;

        x1 = (((int32_t) bmp085.current_temp - (int32_t) bmp085.cal_param.ac6)
            * (int32_t) bmp085.cal_param.ac5) >> 15;
        x2 = ((int32_t) bmp085.cal_param.mc << 11) / (x1 + bmp085.cal_param.md)
            ;
        bmp085.param_b5 = x1 + x2;
```

```
        temperature = ((bmp085.param_b5 + 8) >> 4);  // temperature in 0.1ï¿½C

        return (temperature);
}

// function to convert and scale pressure values to Pascal (taken from the
   BMP085 API from BOSCH)
int32_t bmp085_get_pressure(void)
{
   int32_t pressure,x1,x2,x3,b3,b6;
   uint32_t b4, b7;

   b6 = bmp085.param_b5 - 4000;
   //*****calculate B3************
   x1 = (b6*b6) >> 12;
   x1 *= bmp085.cal_param.b2;
   x1 >>=11;

   x2 = (bmp085.cal_param.ac2*b6);
   x2 >>=11;

   x3 = x1 +x2;

   b3 = (((((int32_t)bmp085.cal_param.ac1 )*4 + x3) <<((int32_t)bmp085.
       oversampling_setting)) + 2) >> 2;

   //*****calculate B4************
   x1 = (bmp085.cal_param.ac3* b6) >> 13;
   x2 = (bmp085.cal_param.b1 * ((b6*b6) >> 12) ) >> 16;
   x3 = ((x1 + x2) + 2) >> 2;
   b4 = (bmp085.cal_param.ac4 * (uint32_t) (x3 + 32768)) >> 15;

   b7 = ((uint32_t)(bmp085.current_pressure - b3) * (50000>>(int32_t)bmp085.
       oversampling_setting));
   if (b7 < 0x80000000)
   {
     pressure = (b7 << 1) / b4;
   }
   else
   {
     pressure = (b7 / b4) << 1;
   }

   x1 = pressure >> 8;
   x1 *= x1;
   x1 = (x1 * BMP085_PARAM_MG) >> 16;
   x2 = (pressure * BMP085_PARAM_MH) >> 16;
   pressure += (x1 + x2 + BMP085_PARAM_MI) >> 4;        // pressure in Pa

   return (pressure);
}


#ifdef IMU_PIXHAWK_V210          // can not be used on Pixhawk v2.0 IMU
/* Interrupt handler for BMP085 End-of-Conversion (EOC) interrupt */
static void EOC_EXTINT_ISR(void) {
```

```
        ISR_ENTRY();                                    // enter ISR

        bmp085_start_measurement_read();                // invoke function
            bmp085_start_measurement_read()

        /* clear EINT */
        EXTINT|=  1 << BMP085_EOC_EINT;

        VICVectAddr = 0x00000000;           /* clear this interrupt from the VIC */
        ISR_EXIT();                                         // exit ISR
}
#endif

#endif
```

C.5. I²C EEPROM Device Driver

C.5.1. eeprom.h

```
/*=========================================================================
PIXHAWK Micro Air Vehicle Flying Robotics Toolkit

(c) 2009, 2010 PIXHAWK PROJECT  <http://pixhawk.ethz.ch>

This file is part of the PIXHAWK project

    PIXHAWK is free software: you can redistribute it and/or modify
    it under the terms of the GNU General Public License as published by
    the Free Software Foundation, either version 3 of the License, or
    (at your option) any later version.

    PIXHAWK is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
    GNU General Public License for more details.

    You should have received a copy of the GNU General Public License
    along with PIXHAWK. If not, see <http://www.gnu.org/licenses/>.

=========================================================================*/

/**
 * @file
 *   @brief Microchip 24FC256 EEPROM driver
 *
 *   This file contains the device driver for the Microchip 24FC256 EEPROM.
 *   @author Christian Dobler <mavteam@student.ethz.ch>
 *
 */
#ifndef EEPROM_H_
#define EEPROM_H_

#include "inttypes.h"
#include "conf.h"
#include "i2c.h"

#ifdef IMU_PIXHAWK_V210

/**
 * @brief Write data to specific address in EEPROM
 *
 * This function writes the "data" byte array to the specified "address"
 * in the EEPROM.
 *
 * @param address        16 byte start address of the location in the EEPROM,
    where the data should be written
 * @param length         number of bytes in the "data" byte array (max. 64)
 * @param data           pointer to the byte array, that will be written to the
    EEPROM
 */
void eeprom_write(uint16_t address, uint8_t length, uint8_t * data);
```

```
/**
 * @brief Start data read from specific address in EEPROM
 *
 * This function starts a read-out from the specified address in the EEPROM.
 * This function does not return any data. "eeprom_read_data()" needs to be
 * called after invoking this function to collect the data read out from the
 * EEPROM. Since it will take a few microseconds to read out the data from
 * EEPROM, other tasks can be executed before collecting the read results,
 * but no other EEPROM requests should be started during this period.
 *
 * @param address        16 byte start address of the location in the EEPROM to
     read from
 * @param length         number of bytes to be read from the EEPROM (max.
     MAX_I2C_PACKAGE_SIZE)
 */
void eeprom_start_read(uint16_t address, uint8_t length);

/**
 * @brief Collect read out data
 *
 * This function collects the data read out from the EEPROM by "
     eeprom_start_read()"
 * in the byte array "data". It has to be called after "eeprom_start_read()"
     and
 * waits for the completion of the EEPROM read by the I2C subsystem. This
     function
 * returns when all the data read from the EEPROM is in the byte array "data".
 *
 * @param data           pointer to the byte array, that the data read from the
     EEPROM is written to
 */
void eeprom_read_data(uint8_t * data);

#endif

#endif /* EEPROM_H_ */
```

C.5.2. eeprom.c

```
/*=====================================================================

PIXHAWK Micro Air Vehicle Flying Robotics Toolkit

(c) 2009, 2010 PIXHAWK PROJECT  <http://pixhawk.ethz.ch>

This file is part of the PIXHAWK project

    PIXHAWK is free software: you can redistribute it and/or modify
    it under the terms of the GNU General Public License as published by
    the Free Software Foundation, either version 3 of the License, or
    (at your option) any later version.

    PIXHAWK is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
```

```
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
    GNU General Public License for more details.

    You should have received a copy of the GNU General Public License
    along with PIXHAWK. If not, see <http://www.gnu.org/licenses/>.

==========================================================================*/

/**
 * @file
 *   @brief Microchip 24FC256 EEPROM driver
 *
 *   This file contains the device driver for the Microchip 24FC256 EEPROM.
 *   @author Christian Dobler <mavteam@student.ethz.ch>
 *
 */

#include "inttypes.h"
#include "eeprom.h"
#include "conf.h"
#include "i2c.h"

#include "comm.h"
#include "led.h"
#include "generated/messages.h"
#include <stdio.h>

#ifdef IMU_PIXHAWK_V210

uint8_t data_buffer[MAX_I2C_PACKAGE_SIZE];
uint8_t data_ready = 0;

/**
 * @brief Temporarily saves read results within the EEPROM driver
 *
 * This function is invoked by the I2C subsystem upon completion of the EEPROM
    read.
 * It stores the read data temporarily within the EEPROM driver. The user
    application
 * can extract this data from the driver by calling "eeprom_read_data()".
 *
 * @param package       I2C package as described in i2c.h
 */
static void eeprom_save_data_after_read(i2c_package *package);


void eeprom_write(uint16_t address, uint8_t length, uint8_t * data)
{
        uint8_t copy_counter;

        // check that byte array is not longer than 64 bytes (max. page write
            length)
        if(length>64) {
                uint8_t buffer[200];     // string buffer for debug messages
                sprintf((char *)buffer, "EEPROM error: Maximum page write size
                    is 64 bytes\n");
                message_send_debug(COMM_0, buffer);
```

```
                return;
        }

        // set up I2C package to be passed to the I2C subsystem
        i2c_package package;
        package.data[1] = (uint8_t)address;
                        // LSB of write address within the EEPROM
        package.data[0] = (uint8_t)(address>>8);                        // MSB
            of write address within the EEPROM
        package.length = length+2;
                                // data bytes + 2 address bytes
        package.direction = I2C_WRITE;
                        // I2C write operation
        package.slave_address = EEPROM_I2C_SLAVE_ADDRESS;        // I2C slave
            address of EEPROM
        package.bus_number = EEPROM_I2C_BUS_NUMBER;                     //
            number of the I2C bus, that the EEPROM is connected to
        package.write_read = 0;
                                // no repeated start condition
        package.i2c_done_handler = NULL;
                        // nothing to be done at I2C completion

        // copy user data to I2C package
        for(copy_counter=0; copy_counter<length; copy_counter++)
        {
                package.data[copy_counter+2] = data[copy_counter];
        }

        i2c_op(&package);        // start I2C write operation
}


void eeprom_start_read(uint16_t address, uint8_t length)
{
        i2c_package package_write, package_read;

        // check that byte array is not longer than MAX_I2C_PACKAGE_SIZE bytes
            (max. transfer size of I2C subsystem)
        if(length>MAX_I2C_PACKAGE_SIZE) {
                uint8_t buffer[200];    // string buffer for debug messages
                sprintf((char *)buffer, "EEPROM error: too large I2C package
                    size specified\n");
                message_send_debug(COMM_0, buffer);
                return;
        }

        // set up I2C package for the read command to be passed to the I2C
            subsystem
        package_write.data[1] = (uint8_t)address;
                                        // LSB of read address within the EEPROM
        package_write.data[0] = (uint8_t)(address>>8);
                                // MSB of read address within the EEPROM
        package_write.length = 2;
                                                // 2 address bytes
        package_write.direction = I2C_WRITE;
                                        // I2C write operation
```

```
        package_write.slave_address = EEPROM_I2C_SLAVE_ADDRESS;
                    // I2C slave address of EEPROM
        package_write.bus_number = EEPROM_I2C_BUS_NUMBER;
                            // number of the I2C bus, that the EEPROM is
            connected to
        package_write.write_read = 1;
                                        // repeated start condition at
            completion
        package_write.i2c_done_handler = NULL;
                                    // nothing to be done at I2C completion

        // set up I2C package for the data read-out to be passed to the I2C
            subsystem
        package_read.length = length;
                                        // number of bytes to be read
        package_read.direction = I2C_READ;
                                        // I2C read operation
        package_read.slave_address = EEPROM_I2C_SLAVE_ADDRESS;
                    // I2C slave address of EEPROM
        package_read.bus_number = EEPROM_I2C_BUS_NUMBER;
                            // number of the I2C bus, that the EEPROM is
            connected to
        package_read.write_read = 1;
                                        // repeated start condition at start
        package_read.i2c_done_handler = &eeprom_save_data_after_read;   //
            eeprom_save_data_after_read() is called at I2C completion

        i2c_write_read(&package_write, &package_read);  // start data read
            operation

}

static void eeprom_save_data_after_read(i2c_package *package)
{
        uint8_t copy_counter;

        // copy data from the I2C package passed by the I2C subsystem to the
            temporary data buffer within this driver
        for(copy_counter=0; copy_counter<package->length; copy_counter++)
        {
                data_buffer[copy_counter] = package->data[copy_counter];
        }

        // set data ready to the number of valid bytes in the data buffer
        data_ready = package->length;
}

void eeprom_read_data(uint8_t * data)
{
        uint8_t copy_counter;

        // wait for data to be read from the EEPROM
        while(!data_ready) ;

        // copy the read data to the byte attay "data" specified by the user
            application
        for(copy_counter=0; copy_counter<data_ready; copy_counter++)
```

```
                {
                        data[copy_counter] = data_buffer[copy_counter];
                }

        // reset data_ready
        data_ready = 0;
}

#endif
```

C.6. I²C Motor Controller Device Driver

C.6.1. i2c_motor_controller.h

```
/*=====================================================================

PIXHAWK Micro Air Vehicle Flying Robotics Toolkit

(c) 2009, 2010 PIXHAWK PROJECT  <http://pixhawk.ethz.ch>

This file is part of the PIXHAWK project

    PIXHAWK is free software: you can redistribute it and/or modify
    it under the terms of the GNU General Public License as published by
    the Free Software Foundation, either version 3 of the License, or
    (at your option) any later version.

    PIXHAWK is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
    GNU General Public License for more details.

    You should have received a copy of the GNU General Public License
    along with PIXHAWK. If not, see <http://www.gnu.org/licenses/>.

=====================================================================*/

/**
 * @file
 *   @brief I2C motor controller driver
 *
 *   This file contains the device driver for the I2C motor controllers.
 *   @author Christian Dobler <mavteam@student.ethz.ch>
 *
 */

#ifndef I2C_MOTOR_CONTROLLER_H_
#define I2C_MOTOR_CONTROLLER_H_

#include "inttypes.h"

#define MOT_GET_RPM_CMD                    0x11     ///< command to read RPM from
    motor controllers (0b00010001)

/**
 * @brief Function to set RPM value on one of the motors
 *
 * This function sends the RPM value specified in rpm to the motor
 * controller specified by mot_i2c_dev_addr.
 *
 * @param mot_i2c_dev_addr I2C slave address of a motor controller. Possible
     values are: MOT1_I2C_SLAVE_ADDRESS, MOT2_I2C_SLAVE_ADDRESS,
     MOT3_I2C_SLAVE_ADDRESS or MOT4_I2C_SLAVE_ADDRESS
 * @param rpm   RPM value to be set
 */
void motor_i2c_set_rpm(uint8_t mot_i2c_dev_addr, uint16_t rpm);
```

```
/**
 * @brief Function to read current RPM value from one of the motors
 *
 * This function reads the current RPM value of the motor
 * controller specified by mot_i2c_dev_addr.
 *
 * @param mot_i2c_dev_addr I2C slave address of a motor controller. Possible
     values are: MOT1_I2C_SLAVE_ADDRESS, MOT2_I2C_SLAVE_ADDRESS,
     MOT3_I2C_SLAVE_ADDRESS or MOT4_I2C_SLAVE_ADDRESS
 * @return RPM value to be set
 */
uint16_t motor_i2c_get_rpm(uint16_t mot_i2c_dev_addr);

#endif /* I2C_MOTOR_CONTROLLER_H_ */
```

C.6.2. i2c_motor_controller.c

```
/*=========================================================================

PIXHAWK Micro Air Vehicle Flying Robotics Toolkit

(c) 2009, 2010 PIXHAWK PROJECT  <http://pixhawk.ethz.ch>

This file is part of the PIXHAWK project

    PIXHAWK is free software: you can redistribute it and/or modify
    it under the terms of the GNU General Public License as published by
    the Free Software Foundation, either version 3 of the License, or
    (at your option) any later version.

    PIXHAWK is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
    GNU General Public License for more details.

    You should have received a copy of the GNU General Public License
    along with PIXHAWK. If not, see <http://www.gnu.org/licenses/>.

=========================================================================*/

/**
 * @file
 *   @brief I2C motor controller driver
 *
 *   This file contains the device driver for the I2C motor controllers.
 *   @author Christian Dobler <mavteam@student.ethz.ch>
 *
 */

#include "inttypes.h"
#include "i2c_motor_controller.h"
#include "conf.h"
#include "i2c.h"

#include "comm.h"
```

```
#include "led.h"
#include "generated/messages.h"
#include <stdio.h>

#if defined(IMU_PIXHAWK_V200) || defined(IMU_PIXHAWK_V210)

static void mot_save_rpm_after_read(i2c_package *package);
volatile uint16_t current_rpm = 0;
volatile uint8_t mot_rpm_data_ready = 0;

void motor_i2c_set_rpm(uint8_t mot_i2c_dev_addr, uint16_t rpm)
{
        i2c_package package;
        package.data[1] = (uint8_t)rpm;
                    // LSB of RPM value to be set
        package.data[0] = (uint8_t)(rpm>>8);                          // MSB
            of RPM value to be set
        package.length = 2;
                                // 2 bytes for RPM setting
        package.direction = I2C_WRITE;
                    // I2C write operation
        package.slave_address = mot_i2c_dev_addr;                     // I2C
            slave address of selected motor controller
        package.bus_number = MOT_I2C_BUS_NUMBER;                      //
            number of the I2C bus, that the motor controller is connected to
        package.write_read = 0;
                        // no repeated start condition
        package.i2c_done_handler = NULL;
                    // nothing to be done at I2C completion
        i2c_op(&package);
}

uint16_t motor_i2c_get_rpm(uint16_t mot_i2c_dev_addr)
{
        i2c_package package_write, package_read;

        // set up I2C package for the read command to be passed to the I2C
            subsystem
        package_write.data[0] = MOT_GET_RPM_CMD;
                        // command to read RPM from motor controller
        package_write.length = 1;
                                    // 1 command byte
        package_write.direction = I2C_WRITE;
                        // I2C write operation
        package_write.slave_address = mot_i2c_dev_addr;
                    // I2C slave address of selected motor controller
        package_write.bus_number = MOT_I2C_BUS_NUMBER;
                    // number of the I2C bus, that the motor controller is
            connected to
        package_write.write_read = 1;
                                // repeated start condition at completion
        package_write.i2c_done_handler = NULL;
                        // nothing to be done at I2C completion

        // set up I2C package for the data read-out to be passed to the I2C
            subsystem
```

```
        package_read.length = 2;
                                        // 2 bytes for MSB and LSB of
            current RPM value to be read
        package_read.direction = I2C_READ;
                                // I2C read operation
        package_read.slave_address = mot_i2c_dev_addr;
                    // I2C slave address of selected motor controller
        package_read.bus_number = MOT_I2C_BUS_NUMBER;
                    // number of the I2C bus, that the motor controller is
            connected to
        package_read.write_read = 1;
                                // repeated start condition at start
        package_read.i2c_done_handler = &mot_save_rpm_after_read;       //
            mot_save_rpm_after_read() is called at I2C completion

        mot_rpm_data_ready = 0;
                                        // invalidate old RPM value

        i2c_write_read(&package_write, &package_read);
                    // start RPM read operation

        while(!mot_rpm_data_ready) ;
                                // wait for I2C operation to complete

        return current_rpm;
                                                // return current RPM value
}

static void mot_save_rpm_after_read(i2c_package *package)
{
        current_rpm = (((uint16_t)(package->data[0]))<<8) || ((uint16_t)(
            package->data[1]));     // temporarily store RPM value contained in
            the I2C package to current_rpm
        mot_rpm_data_ready = 1;

                                // set new RPM value to be valid
}

#endif
```

C.7. SPI Subsystem

C.7.1. spi.h

```
/**
* @file spi.h
*
* @brief functions to use spi
*
* This file contains different functions to use spi interface of
* the arm7. It uses the ssp port and select then the spi function
* of the ssp. The spi functions are optimized for reading out sensor
* data.
**/

#ifndef SPI_H_
#define SPI_H_

#include "LPC21xx.h"

static inline void SpiEnable(void) {
        SSPCR1 |= (1 << SSE);
}
static inline void SpiDisable(void) {
        SSPCR1 &= ~(1 << SSE);
}
static inline void SpiEnableRti(void) {
        SSPIMSC |= (1 << RTIM);
}
static inline void SpiDisableRti(void) {
        SSPIMSC &= ~(1 << RTIM);
}
static inline void SpiClearRti(void) {
        SSPICR |= (1 << RTIC);
}

/*! Constant for selecting the spi 5 bit_mode. @see spi_package */
#define SPI_5_BIT_MODE  0x04 << 0
/*! Constant for selecting the spi 8 bit_mode. @see spi_package */
#define SPI_8_BIT_MODE  0x07 << 0
/*! Constant for selecting the spi 16 bit_mode. @see spi_package */
#define SPI_16_BIT_MODE 0x0F << 0

/**
 * @brief struct of data, mode and some spi functions.
 *
 * If you want to communicate with an spi device, you have first
 * to create such a struct. In this struct there is not only data,
 * but also the slave select functions for your device and the spi
 * interrupt handler for your device.
 * @see spi_transmit
 */
typedef struct {
        /*! An array of max 8 frames. You can put in here 8 or 16 bit
         * values. You have then to choose the correct bit_mode */
        unsigned short data[8];
        /*! the number of frames in the data array */
```

```
        unsigned char length;
        /*! The number of bits for one spi transfer. You can
         * choose here SPI_8_BIT_MODE or SPI_16_BIT_MODE
         * @see SPI_8_BIT_MODE
         * @see SPI_16_BIT_MODE */
        unsigned char bit_mode;
        /*! The slave select function for your spi device. You have to
         * set the slave select pin of your device to low to be able to
         * communicate with it. */
        void (*slave_select)(void);
        /*! The slave deselect function for your spi device. You have to
         * set the slave select pin of your device to high if you don't
         * communicate with it, if you don't do this, other spi devices
         * might have problems. */
        void (*slave_unselect)(void);
        /*! After the transmission of the data of your spi package is
         * finished, an spi receive timeout interrupt will occur. The
         * SPI_ISR will call this interrupt handler. In this interrupt
         * handler you have to read out dummy frames for data you have
         * sent or you have to read out the data which you have ordered */
        void (*spi_interrupt_handler)(void);
} spi_package;

/**
 * @brief spi initialization
 *
 * Initialize the spi bus. If you want to use any spi device, you first
 * have to use this initialization function.
 */
void spi_init(void);

/**
 * @brief send and receive data over spi
 *
 * Transmit a spi package over the spi bus. If you want to receive something
 * over spi, you have to send 0 over spi and handle it with your
 * spi_interrupt_handler. Also if you want to just transmit a file you
 * have to read out a dummy frame out of the spi buffer.
 * @param package The package which you want to send over spi. In the package
 * is also the spi_interrupt_handler and the slave select and unselect
     functions.
 * @see spi_package
 */
void spi_transmit( spi_package* package );

int spi_running(void);

/**
 * With this function you can check how many packages are in the spi buffer.
 * @return number of packages in the spi buffer
 */
int spi_number_of_packages_in_buffer(void);
#endif /* SPI_H_ */
```

C.7.2. spi.c

```
/*
 * spi.c
 *
 *  Created on: 15.05.2009
 *      Author: Martin Rutschmann
 */

#include "spi.h"
#include "armVIC.h"
#include "LPC21xx.h"
#include "conf.h"
#include "led.h"

#include <stdio.h>

#ifdef SPI_USE_POLLING

void spi_transmit(spi_package* package) {
        /*copy data into fifo*/
        for(int i=0; i<(*package).length; i++){
                SSPDR = (*package).data[i];
        }
        /*set bit mode (8bit or 16bit)*/
        SSPCR0 = (*package).bit_mode | SSP_FRF | SSP_CPOL | SSP_CPHA | SSP_SCR
            ;
        /*select the device*/
        (*package).slave_select();
        /*enable spi and send fifo*/
        SpiEnable();
        /*wait until all data has been sent(Transmit fifo is emtpy)*/
        while(SSPSR&(1<<BSY));
        /*unselect device*/
        (*package).slave_unselect();
        /*disable spi*/
        SpiDisable();
        /*read out the dummy bytes or the data bytes*/
        (*package).spi_interrupt_handler();
}

void spi_init(void){
        /* setup pins for SSP (SCK, MISO, MOSI) */
        PINSEL1 |= SSP_PINSEL1_SCK | SSP_PINSEL1_MISO | SSP_PINSEL1_MOSI;

        /* setup SSP */
        SSPCR0 = SSPCR0_VAL;
        SSPCR1 = SSPCR1_VAL;
        SSPCPSR = 0x02;
}

int spi_running(void){
        return 0;
}
```

```
int spi_number_of_packages_in_buffer(void){
        //in polling mode the buffer isn't needed
        return 0;
}

#else

static void SSP_ISR(void) __attribute__((naked));
static inline void spi_transmit_single_package(spi_package* package);

spi_package spi_package_buffer[SPI_PACKAGE_BUFFER_SIZE];
int spi_package_buffer_insert_idx, spi_package_buffer_extract_idx;
int spi_transmit_running;

spi_package* spi_current_package;

void spi_transmit(spi_package* package) {
        int temp;
        unsigned cpsr;

        temp = (spi_package_buffer_insert_idx + 1) % SPI_PACKAGE_BUFFER_SIZE;
            // calculate the next queue position

        if (temp == spi_package_buffer_extract_idx) { // check if there is free
             space in the send queue
                return; // no room
        }

        cpsr = disableIRQ(); // disable global interrupts
        SpiDisableRti(); // disable RTI interrupts
        restoreIRQ(cpsr); // restore global interrupts

        spi_package_buffer[spi_package_buffer_insert_idx] = *package; // add
            data to queue
        spi_package_buffer_insert_idx = temp; // increase insert pointer

        if (spi_transmit_running==0) // check if in process of sending data
        {
                spi_transmit_running = 1; // set running flag
                spi_transmit_single_package(&spi_package_buffer[
                    spi_package_buffer_extract_idx]);
                spi_package_buffer_extract_idx++;
                spi_package_buffer_extract_idx %= SPI_PACKAGE_BUFFER_SIZE;
        }

        cpsr = disableIRQ(); // disable global interrupts
        SpiEnableRti(); // enable RTI interrupts
        restoreIRQ(cpsr); // restore global interrupts
}

void spi_init(void){
        /* setup pins for SSP (SCK, MISO, MOSI) */
        PINSEL1 |= SSP_PINSEL1_SCK  | SSP_PINSEL1_MISO | SSP_PINSEL1_MOSI;

        /* setup SSP */
        SSPCR0 = SSPCR0_VAL;
```

```
        SSPCR1 = SSPCR1_VAL;
        SSPCPSR = 0x02;

        /* initialize interrupt vector */
        VICIntSelect &= ~VIC_BIT( VIC_SPI1 );  /* SPI1 selected as IRQ */
        VICIntEnable = VIC_BIT( VIC_SPI1 );    /* enable it            */
        _VIC_CNTL(SSP_VIC_SLOT) = VIC_ENABLE | VIC_SPI1;
        _VIC_ADDR(SSP_VIC_SLOT) = (unsigned int)SSP_ISR;      /* address of the
            ISR   */

        spi_transmit_running = 0;
        spi_package_buffer_insert_idx = 0;
        spi_package_buffer_extract_idx = 0;
}

static inline void spi_transmit_single_package(spi_package* package){
        spi_current_package = package;
        /*copy data into fifo*/
        for(int i=0; i<(*spi_current_package).length; i++){
                SSPDR = (*spi_current_package).data[i];
        }
        /*set bit mode (8bit or 16bit)*/
        SSPCR0 = (*spi_current_package).bit_mode |  SSP_FRF | SSP_CPOL |
            SSP_CPHA | SSP_SCR;
        /*select the device*/
        (*spi_current_package).slave_select();

        /* enable SPI and send package data*/
        SpiClearRti();
        SpiEnableRti();
        SpiEnable();
}

int spi_number_of_packages_in_buffer(void){
        return((spi_package_buffer_insert_idx-spi_package_buffer_extract_idx) %
            SPI_PACKAGE_BUFFER_SIZE);
}

int spi_running(void){
        return spi_transmit_running;
}

static void SSP_ISR(void) {
        ISR_ENTRY();
        //start the device interrupt handler
        (*spi_current_package).spi_interrupt_handler();

        //unselect device, disable spi
        (*spi_current_package).slave_unselect();
        SpiClearRti();
        SpiDisableRti();
        SpiDisable();

        // check if more data to send
        if (spi_package_buffer_insert_idx != spi_package_buffer_extract_idx) {
                spi_transmit_single_package(&spi_package_buffer[
                    spi_package_buffer_extract_idx]);
```

```
                spi_package_buffer_extract_idx++;
                spi_package_buffer_extract_idx %= SPI_PACKAGE_BUFFER_SIZE;
        } else {
                spi_transmit_running = 0; // clear running flag
        }
        VICVectAddr = 0x00000000; /* clear this interrupt from the VIC */
        ISR_EXIT();
}

#endif
```

C.8. SCA3100 3D Accelerometer Device Driver

C.8.1. sca3100.h

```
/*=========================================================================
PIXHAWK Micro Air Vehicle Flying Robotics Toolkit

(c) 2009, 2010 PIXHAWK PROJECT  <http://pixhawk.ethz.ch>

This file is part of the PIXHAWK project

    PIXHAWK is free software: you can redistribute it and/or modify
    it under the terms of the GNU General Public License as published by
    the Free Software Foundation, either version 3 of the License, or
    (at your option) any later version.

    PIXHAWK is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
    GNU General Public License for more details.

    You should have received a copy of the GNU General Public License
    along with PIXHAWK. If not, see <http://www.gnu.org/licenses/>.

=========================================================================*/

/**
 * @file
 *   @brief SCA3100 linear accelerometer device driver
 *
 *   This file contains the device driver for the VTI SCA3100 linear
     accelerometer.
 *   @author Christian Dobler <mavteam@student.ethz.ch>
 *
 */


#ifndef SCA3100_H_
#define SCA3100_H_

#define SCA3100_MAX_NEG_VALUE    8192            ///< max. neg. value of sensor
    for two's complement calculation

/**
 * @brief Initialization of the SCA3100 linear accelerometer
 *
 * This function initializes the SCA3100 linear accelerometer and sets the chip
 * select pin into operational mode. The SPI subsystem has to be initialized
 * beforehand; otherwise this will fail.
 */
void sca3100_init(void);

/**
 * @brief Starts reading of current acceleration measurements
 *
```

```
 * This function starts the read operation of the SCA3100 sensor values over
     the
 * SPI bus. The SPI subsystem will complete the data transfer and the
     measurements
 * are temporarily saved in the SCA3100 driver. Execute "while (spi_running())
     ;"
 * to wait for the results. Then call sca3100_get_value() to retrieve
 * the measurements read out of the sensor.
 */
void sca3100_read_res(void);

/**
 * @brief Retrieves most recent measurement from SCA3100 device driver
 *
 * This function retrieves the most recent measurement from SCA3100 device
     driver.
 * To read out the current values from the sensor, execute sca3100_read_res()
     first.
 */
int sca3100_get_value(int axis);

/**
 * @brief Resets the PORST bit
 *
 * This function is used to reset the PORST (power-on-reset) bit in the status
     byte
 * transmitted by the SCA3100. This bit is set when the sensor has been reset
     and needs
 * to be cleared. Call this function once after reset of the MCU as soon as SPI
     , interrupts
 * and timers are running. If these components are not running, this function
     will fail.
 */
void sca3100_reset_porst_bit(void);

/**
 * @brief Resets the PORST bit
 *
 * This function is used to reset the INT-STATUS register to the normal state.
 * Call this function once after reset of the MCU as soon as SPI, interrupts
     and
 * timers are running. If these components are not running, this function will
     fail.
 */
void sca3100_read_int_status(void);


#endif /* SCA3100_H_ */
```

C.8.2. sca3100.c

```
/*==========================================================================

PIXHAWK Micro Air Vehicle Flying Robotics Toolkit
```

```
(c) 2009, 2010 PIXHAWK PROJECT  <http://pixhawk.ethz.ch>

This file is part of the PIXHAWK project

    PIXHAWK is free software: you can redistribute it and/or modify
    it under the terms of the GNU General Public License as published by
    the Free Software Foundation, either version 3 of the License, or
    (at your option) any later version.

    PIXHAWK is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
    GNU General Public License for more details.

    You should have received a copy of the GNU General Public License
    along with PIXHAWK. If not, see <http://www.gnu.org/licenses/>.

==========================================================================*/

/**
 * @file
 *   @brief SCA3100 linear accelerometer device driver
 *
 *   This file contains the device driver for the VTI SCA3100 linear
     accelerometer.
 *   @author Christian Dobler <mavteam@student.ethz.ch>
 *
 */

#include "spi.h"
#include "sca3100.h"
#include "armVIC.h"
#include "conf.h"
#include "comm.h"
#include "led.h"
#include "generated/messages.h"
#include "inttypes.h"
#include <stdio.h>


#if defined(IMU_PIXHAWK_V200) || defined(IMU_PIXHAWK_V210)

int sca3100_values[3];                                          // temporary
    storage for measured acceleration values

void sca3100_spi_init(void);

static void sca3100_select(void);
static void sca3100_unselect(void);
static void sca3100_on_spi_int(void);
static void sca3100_on_spi_read_reg(void);
static void sca3100_on_spi_write(void);

static void sca3100_select(void){                           // pull SCA3100 chip
    select low
        SCA3100_SS_IOCLR|=1<<SCA3100_SS_PIN;
```

```
}
static void sca3100_unselect(void){                    // pull SCA3100 chip
    select high
        SCA3100_SS_IOSET|=1<<SCA3100_SS_PIN;
}


void sca3100_init(void) {
        /* configure SS pin */
        SCA3100_SS_IODIR|=1<<SCA3100_SS_PIN;     /* pin is output  */
        sca3100_unselect();                                    /* pin idles
            high */

        //spi_package package;

        /* set PORST bit to 0 */
        /*package.data[0] = 0x07;
        package.data[1] = 0x00;
        package.length = 2;
        package.slave_select = &sca3100_select;
        package.slave_unselect = &sca3100_unselect;
        package.spi_interrupt_handler = &sca3100_on_spi_int;
        spi_transmit(&package);*/

        //led_on(LED_GREEN);

        // set all measurement results to 0
        for(int i=0; i<3; i++){
                sca3100_values[i]=0;
        }
        sca3100_reset_porst_bit();
        sca3100_read_int_status();
}


// start a measurement read operation
void sca3100_read_res(void) {

    spi_package package;
        package.bit_mode = SPI_8_BIT_MODE;

        /* trigger 2 bytes read */
    unsigned char cmd = 0x15;            // command byte: start with reading
        axis, address counter increments automatically
        package.data[0] = cmd;                   // send command byte first
        package.data[1] = 0;
        package.data[2] = 0;
        package.data[3] = 0;
        package.data[4] = 0;
        package.data[5] = 0;
        package.data[6] = 0;
        package.length = 7;                                    // send 1 command byte
            and read 6 data bytes
        package.slave_select = &sca3100_select;
        package.slave_unselect = &sca3100_unselect;
        package.spi_interrupt_handler = &sca3100_on_spi_int;     //
            sca3100_on_spi_int() is called at SPI completion
```

```
        spi_transmit(&package);                 // start SPI operation
}

static void sca3100_on_spi_int(void) {


        /* read dummy control byte reply */
        unsigned char sca3100_status = SSPDR;   // temporarily save received
            SCA3100 status byte (first byte received)
        uint8_t buffer[200];
        spi_package package;
        package.bit_mode = SPI_8_BIT_MODE;
        unsigned char cmd;
        unsigned char cmd_val;
        unsigned char val_ok = 1;       // start with val_ok=1, if anything
            went wrong, val_ok will be set to 0 and new values discarded

        if((sca3100_status & 0x7C) || !(sca3100_status & (1<<1))) {

                // SPI frame error in previously sent frame was detected
                if(sca3100_status & (1<<6)) {
                        sprintf((char *)buffer, "FRME: SPI frame error (
                            previous frame)\n");
                        message_send_debug(COMM_1, buffer);
                }

                // sensor has been reset since the last measurement
                if(sca3100_status & (1<<5)) {
                        sprintf((char *)buffer, "PORST: unexpected sensor reset
                            occurred\n");
                        message_send_debug(COMM_1, buffer);
                        val_ok = 0;

                }

                // one of the device self tests has failed
                if(sca3100_status & (1<<4)) {
                        val_ok = 0;
                        //led_on(LED_GREEN);
                }

                // sensor was in saturation, when measuring the current value
                    -> discard measurement
                if(sca3100_status & (1<<3)) {
                        led_on(LED_GREEN);
                        sprintf((char *)buffer, "SAT: sensor saturation
                            occurred\n");
                        message_send_debug(COMM_1, buffer);
                        val_ok = 0;
                }

                // fixed SCA3100 control bits are received incorrectly
                if((sca3100_status & (1<<2)) || !(sca3100_status & (1<<1))) {
                        sprintf((char *)buffer, "FRME: SPI receive error (
                            current frame)\n");
                        message_send_debug(COMM_1, buffer);
                        val_ok = 0;
```

```
        }

    }

    // if status byte is ok, copy and convert measurement data to mg
    if(val_ok)
    {
        // X axis first:
        unsigned char msb;
        msb = SSPDR;                                    // read MSB
            from SPI data register
        unsigned char lsb = SSPDR;              // read LSB from SPI
            data register
        // Convert 2's complement (Manual page 14, section 3.1.4 Output
             data conversion)
        // Copy msb and lsb to (msb|lsb)
        uint16_t raw_value = ((uint16_t)msb<<8) | lsb;
        int16_t mg_value;
        // Align bit pattern correctly to 16bit, last two bits not
            defined
        raw_value = raw_value>>2;
        // convert raw_value in two's complement to mg_value in normal
            binary format and scale it to mg
        if(raw_value & (1<<13))
        {
                raw_value &= 0x1FFF;
                mg_value = ((int16_t)raw_value-SCA3100_MAX_NEG_VALUE)
                    *(10)/9;
        }
        else {
                mg_value = (int16_t)raw_value*(10)/9;
        }
        sca3100_values[SCA3100_X_AXIS] = (int)mg_value;         //
            store measurement for X axis

        // same for the Z axis:
        msb = SSPDR;                                    // read MSB
            from SPI data register
        lsb = SSPDR;                                    // read LSB
            from SPI data register
        raw_value = ((uint16_t)msb<<8) | lsb;
        raw_value = raw_value>>2;
        // convert raw_value in two's complement to mg_value in normal
            binary format and scale it to mg
        if(raw_value & (1<<13)) {
                raw_value &= 0x1FFF;
                mg_value = ((int16_t)raw_value-SCA3100_MAX_NEG_VALUE)
                    *(10)/9;
        }
        else {
                mg_value = (int16_t)raw_value*(10)/9;
        }
        sca3100_values[SCA3100_Z_AXIS] = (int)mg_value;         //
            store measurement for Z axis

        // same for the Y axis:
```

```
            msb = SSPDR;                                    // read MSB
                from SPI data register
            lsb = SSPDR;                                    // read LSB
                from SPI data register
            raw_value = ((uint16_t)msb<<8) | lsb;
            raw_value = raw_value>>2;
            // convert raw_value in two's complement to mg_value in normal
                binary format and scale it to mg
            if(raw_value & (1<<13)) {
                    raw_value &= 0x1FFF;
                    mg_value = ((int16_t)raw_value-SCA3100_MAX_NEG_VALUE)
                        *(10)/9;
            }
            else {
                    mg_value = (int16_t)raw_value*(10)/9;
            }
            sca3100_values[SCA3100_Y_AXIS] = (int)mg_value;         //
                store measurement for Y axis

    }

    // discard received data, if status byte from SCA3100 was not ok
    else {
            int i;
            unsigned char foo __attribute__ ((unused));
            for(i=0; i<6;i++) {
                    foo = SSPDR;
            }
    }

    // if anything in the status byte was abnormal, do the following
    if(sca3100_status & 0x7C) {

            // if the PORST bit was set, issue the write command 0x0700 to
                clear it (not supposed to happen under normal operation)
            if(sca3100_status & (1<<5)) {
                    cmd = 0x07;
                    cmd_val = 0x00;
                    package.data[0] = cmd;
                    package.data[1] = cmd_val;
                    package.length = 2;
                    package.slave_select = &sca3100_select;
                    package.slave_unselect = &sca3100_unselect;
                    package.spi_interrupt_handler = &sca3100_on_spi_write;
                          // sca3100_on_spi_write() is invoked at SPI
                        completion
                    spi_transmit(&package);
                    led_on(LED_GREEN);
                    return;
            }


            // if the SAT or ST bit was set, read the INT_STATUS register
                to determine exact cause and clear flags (not supposed to
                happen under normal operation)
            if((sca3100_status & (1<<4)) || (sca3100_status & (1<<3))) {
                    led_on(LED_GREEN);
```

```
                        cmd = 0x58;
                                        // 0x5800 is the command to read the
                                status register
                        cmd_val = 0x00;
                                    // all commands: page 23 in SCA3100 family
                            datasheet
                        package.data[0] = cmd;
                        package.data[1] = cmd_val;
                        package.length = 2;
                        package.slave_select = &sca3100_select;
                        package.slave_unselect = &sca3100_unselect;
                        package.spi_interrupt_handler = &
                            sca3100_on_spi_read_reg;        //
                            sca3100_on_spi_read_reg() is invoked at SPI
                            completion
                        spi_transmit(&package);
                        return;
                }

        }

        return;
}


int sca3100_get_value(int axis){
        return sca3100_values[axis];     // return value of specified axis
}

void sca3100_reset_porst_bit(void) {
        spi_package package;
        package.bit_mode = SPI_8_BIT_MODE;
        unsigned char cmd;
        unsigned char cmd_val;
        cmd = 0x07;                                     // 0x0700 is the
            command to reset the PORST bit
        cmd_val = 0x00;
        package.data[0] = cmd;
        package.data[1] = cmd_val;
        package.length = 2;
        package.slave_select = &sca3100_select;
        package.slave_unselect = &sca3100_unselect;
        package.spi_interrupt_handler = &sca3100_on_spi_write;  //
            sca3100_on_spi_write() is invoked at SPI completion
        spi_transmit(&package);
        //led_on(LED_GREEN);
        return;
}

void sca3100_read_int_status(void) {
        spi_package package;
        package.bit_mode = SPI_8_BIT_MODE;
        unsigned char cmd;
        unsigned char cmd_val;
        cmd = 0x58;                                     // 0x5800 is the
            command to read the status register
        cmd_val = 0x00;
```

```
        package.data[0] = cmd;
        package.data[1] = cmd_val;
        package.length = 2;
        package.slave_select = &sca3100_select;
        package.slave_unselect = &sca3100_unselect;
        package.spi_interrupt_handler = &sca3100_on_spi_read_reg;        //
            sca3100_on_spi_read_reg() is invoked at SPI completion
        spi_transmit(&package);
        return;
}

static void sca3100_on_spi_write(void) {
        unsigned char foo __attribute__ ((unused)) = SSPDR;      // this was a
            write operation -> discard any received data (2 bytes)
        foo = SSPDR;
        //led_on(LED_GREEN);
        return;
}

static void sca3100_on_spi_read_reg(void) {              // INT_STATUS register
    has been read
        uint8_t buffer[200];
        spi_package package;
        package.bit_mode = SPI_8_BIT_MODE;
        unsigned char cmd;
        unsigned char cmd_val;
        unsigned char foo __attribute__ ((unused)) = SSPDR;      // discard
            first byte
        unsigned char int_status = SSPDR;
                    // INT_STATUS register content is in the second byte (this
            byte)
        // if STS has failed
        if(int_status & (1<<5)) {
                sprintf((char *)buffer, "STS: Start-up self-test failed\n");
                message_send_debug(COMM_1, buffer);
        }
        // if STC has failed
        if(int_status & (1<<4)) {
                sprintf((char *)buffer, "STC: Continuous self-test failed\n");
                message_send_debug(COMM_1, buffer);
        }
        // if not SAT & not STS & not STC, then memory self test has failed ->
            do another memory self-test
        if(!((int_status & (1<<6)) || (int_status & (1<<4)) || (int_status &
            (1<<5)))) {
                sprintf((char *)buffer, "ST: Memory self test-failed\n");
                message_send_debug(COMM_1, buffer);
                cmd = 0x07;                              // 0x0704 is the
                    command to start a new memory self test
                cmd_val = 0x04;
                package.data[0] = cmd;
                package.data[1] = cmd_val;
                package.length = 2;
                package.slave_select = &sca3100_select;
                package.slave_unselect = &sca3100_unselect;
                package.spi_interrupt_handler = &sca3100_on_spi_write;   //
                    sca3100_on_spi_write() is invoked at SPI completion
```

```
                spi_transmit(&package);
        }
        led_off(LED_GREEN);
        return;
}

#endif
```

C.9. ADS8341 ADC Device Driver

C.9.1. ads8341.h

```
/*=====================================================================

PIXHAWK mavlib - The Micro Air Vehicle Platform Library
Please see our website at <http://pixhawk.ethz.ch>


Original Authors:
  Martin Rutschmann <mavteam@student.ethz.ch>
  Dominik Honegger <mavteam@student.ethz.ch>
  Lorenz Meier <lm@student.ethz.ch>
Contributing Authors (in alphabetical order):


Todo:


(c) 2009 PIXHAWK PROJECT

This file is part of the PIXHAWK project

    mavlib is free software: you can redistribute it and/or modify
    it under the terms of the GNU General Public License as published by
    the Free Software Foundation, either version 3 of the License, or
    (at your option) any later version.

    mavlib is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
    GNU General Public License for more details.

    You should have received a copy of the GNU General Public License
    along with mavlib. If not, see <http://www.gnu.org/licenses/>.

=====================================================================*/

#ifndef ADS8341_H_
#define ADS8341_H_
#include "inttypes.h"

void ads8341_init(void);
uint16_t ads8341_get_value(uint8_t adc_id, uint8_t channel);
void ads8341_read(uint8_t adc_id, uint8_t channel);

#endif /* ADS8341_H_ */
```

C.9.2. ads8341.c

```
/*=====================================================================

PIXHAWK mavlib - The Micro Air Vehicle Platform Library
Please see our website at <http://pixhawk.ethz.ch>
```

```
Original Authors:
  Martin Rutschmann <mavteam@student.ethz.ch>
  Dominik Honegger <mavteam@student.ethz.ch>
  Lorenz Meier <lm@student.ethz.ch>
Contributing Authors (in alphabetical order):


Todo:


(c) 2009 PIXHAWK PROJECT

This file is part of the PIXHAWK project

    mavlib is free software: you can redistribute it and/or modify
    it under the terms of the GNU General Public License as published by
    the Free Software Foundation, either version 3 of the License, or
    (at your option) any later version.

    mavlib is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
    GNU General Public License for more details.

    You should have received a copy of the GNU General Public License
    along with mavlib. If not, see <http://www.gnu.org/licenses/>.

================================================================================*/

#include "ads8341.h"
#include "armVIC.h"
#include "LPC21xx.h"
#include "conf.h"
#include "spi.h"

uint16_t ads8341_value[3][4];
uint8_t ads8341_current_channel;
uint8_t ads8341_current_adc_id;

static void ads8341_unselect(void);
static void ads8341_select(void);
static void ads8341_on_spi_int(void);


void ads8341_init(void)
{
        /* configure SS pin */
        ADS8341_0_SS_IODIR |= 1 << ADS8341_0_SS_PIN; /* pin is output  */
        ads8341_current_adc_id = ADS8341_0;
        ads8341_unselect(); /* pin idles high */

        //ADS8341_1_SS_IODIR |= 1 << ADS8341_1_SS_PIN; /* pin is output  */
        //ads8341_current_adc_id = ADS8341_1;
        //ads8341_unselect(); /* pin idles high */
```

```
        //ADS8341_2_SS_IODIR |= 1 << ADS8341_2_SS_PIN; /* pin is output  */
        //ads8341_current_adc_id = ADS8341_2;
        //ads8341_unselect(); /* pin idles high */
}
static void ads8341_unselect(void)
{
        if (ads8341_current_adc_id == ADS8341_0){
                ADS8341_0_SS_IOSET |= 1 << ADS8341_0_SS_PIN;
        }
        /*case ADS8341_1:
                ADS8341_1_SS_IOSET |= 1 << ADS8341_1_SS_PIN;
                break;
        case ADS8341_2:
                ADS8341_2_SS_IOSET |= 1 << ADS8341_2_SS_PIN;
                break;
        }*/
}
static void ads8341_select(void)
{
        if (ads8341_current_adc_id == ADS8341_0){
                ADS8341_0_SS_IOCLR |= 1 << ADS8341_0_SS_PIN;
        }
        /*case ADS8341_1:
                ADS8341_1_SS_IOCLR |= 1 << ADS8341_1_SS_PIN;
                break;
        case ADS8341_2:
                ADS8341_2_SS_IOCLR |= 1 << ADS8341_2_SS_PIN;
                break;
        }*/
}


void ads8341_read(uint8_t adc_id, uint8_t channel)
{
        ads8341_current_channel=channel;
        ads8341_current_adc_id=adc_id;
        unsigned char cmd1 = (1 << 4);
        switch (channel)
        {
        case 0:
                cmd1 |= (1 << 1);
                break;
        case 1:
                cmd1 |= (1 << 3 | 1 << 1);
                break;
        case 2:
                cmd1 |= (1 << 2);
                break;
        case 3:
                cmd1 |= (1 << 3 | 1 << 2);
                break;
        default:
                return;
        }
        unsigned char cmd2 = (1 << 4) | (1 << 3) | (1 << 2);
        spi_package package;
        package.bit_mode = SPI_5_BIT_MODE;
```

```
        package.data[0] = cmd1;
        package.data[1] = cmd2;
        package.data[2] = 0;
        package.data[3] = 0;
        package.data[4] = 0;
        package.length = 5;
        package.slave_select = &ads8341_select;
        package.slave_unselect = &ads8341_unselect;
        package.spi_interrupt_handler = &ads8341_on_spi_int;
        spi_transmit(&package);

}

static void ads8341_on_spi_int(void)
{
        unsigned char foo __attribute__ ((unused)) = SSPDR;
        unsigned int data = (SSPDR&1)<<15;
        data += SSPDR<<10;
        data += SSPDR<<5;
        data += SSPDR<<0;
        ads8341_value[ads8341_current_adc_id][ads8341_current_channel] = data;
}

// TODO Enable the use of multiple ADCs of the same type
uint16_t ads8341_get_value(uint8_t adc_id, uint8_t channel)
{
        return ads8341_value[adc_id][channel];
}
```

C.10. MS2100 3D Magnetometer Device Driver

C.10.1. ms2100.h

```
/*
 * ms2100.h
 *
 *  Created on: May 14, 2009
 *      Author: mavteam
 */


#ifndef MS2100_H_
#define MS2100_H_

#include "conf.h"
#include "inttypes.h"

#define MS2100_RESET_TIMEOUT_BASE        ((PCLK/10000000UL)?(PCLK/10000000UL):1)

//other functions//////////////////////////////////////////////////////
void ms2100_init(void);
void ms2100_send_request(uint8_t axis);
short ms2100_get_value(int axis);
int ms2100_get_status(void);

#define MS2100_SAMPLING_TIMES   10

#define MS2100_X_AXIS                0
#define MS2100_Y_AXIS            1
#define MS2100_Z_AXIS            2

#define MS2100_IDLE              0
#define MS2100_BUSY              1
#define MS2100_SENDING_REQ       2
#define MS2100_WAITING_EOC       3
#define MS2100_GOT_EOC           4
#define MS2100_READING_RES       5
#define MS2100_DATA_AVAILABLE    6


#endif /* MS2100_H_ */
```

C.10.2. ms2100.c

```
/*
 * ms2100.c
 *
 *  Created on: May 14, 2009
 *      Author: mavteam
 */


#include "spi.h"
#include "ms2100.h"
```

```
#include "armVIC.h"
#include "conf.h"
#include "sys_time.h"

#include "comm.h"
#include "led.h"
#include "generated/messages.h"
#include "inttypes.h"
#include <stdio.h>

static void EXTINT_ISR(void) __attribute__((naked));

/**
 * This function sets the MS2100 pin to low. To reset the sensor you have to
 * first run ms2100_set and then ms2100_reset.
 */
static void ms2100_reset(void);

/**
 * This function sets the MS2100 pin to high. To reset the sensor you have to
 * first run ms2100_set and then ms2100_reset.
 */
static void ms2100_set(void);

/**
 * This function selects the ms2100. If you want to read out a result, then you
 * have to select the ms2100 with this function.
 */
static void ms2100_select(void);

/**
 * This function selects the ms2100 and reset the device. If you want to send
 * a request, you have to choose this select function.
 */
static void ms2100_select_and_reset(void);

/**
 * After a spi transfer is finished, the slave select pin of the ms2100 has to
    go to high.
 */
static void ms2100_unselect(void);

/**
 * spi receive timeout interrupt function for reading.
 */
static void ms2100_on_spi_int_read(void);

/**
 * spi receive timeout interrupt function for a request.
 */
static void ms2100_on_spi_int_req(void);

static void ms2100_do_before_spi_transmit(void);


uint8_t ms2100_status;
uint8_t ms2100_current_axis;
```

```
uint8_t ms2100_sampling_nr;
int16_t ms2100_values[3][MS2100_SAMPLING_TIMES];

spi_package package_read;
spi_package package_req;


static void ms2100_reset(void) {
        MS2100_RESET_IOCLR |= 1 << MS2100_RESET_PIN;//TO LOW
}


static void ms2100_set(void) {
        MS2100_RESET_IOSET |= 1 << MS2100_RESET_PIN;//TO HIGH
}


static void ms2100_select_and_reset(void){
        MS2100_SS_IOCLR |= 1 << MS2100_SS_PIN;
        ms2100_set();
        ms2100_reset();
}


static void ms2100_select(void){
        MS2100_SS_IOCLR|=1<<MS2100_SS_PIN;
}


static void ms2100_unselect(void){
        MS2100_SS_IOSET|=1<<MS2100_SS_PIN;
}


void ms2100_init(void) {
        ms2100_status=MS2100_IDLE;
        for (int i = 0; i < 3; i++) {
                for (int j=0; j<MS2100_SAMPLING_TIMES; j++){
                        ms2100_values[i][j]=0;
                }
        }

        /* configure SS pin */
        MS2100_SS_IODIR|=1<<MS2100_SS_PIN; /* pin is output  */
        ms2100_unselect(); /* pin idles high */

        /* configure RESET pin */
        MS2100_RESET_IODIR|=1<<MS2100_RESET_PIN; /* pin is output  */
        ms2100_reset(); /* pin idles low  */

        /* configure DRDY pin */
        /* connected pin to EXINT */
        MS2100_DRDY_PINSEL |= MS2100_DRDY_PINSEL_VAL << MS2100_DRDY_PINSEL_BIT;
        EXTMODE|= 1 << MS2100_DRDY_EINT; /* EINT is edge trigered */
        EXTPOLAR|= 1 << MS2100_DRDY_EINT; /* EINT is trigered on rising edge */
        EXTINT|=  1 << MS2100_DRDY_EINT; /* clear pending EINT */
```

```
        /* initialize interrupt vector */
        VICIntSelect &= ~VIC_BIT(MS2100_DRDY_VIC_IT); /* select EINT as IRQ
            source */
        VICIntEnable = VIC_BIT(MS2100_DRDY_VIC_IT); /* enable it
                                    */
        _VIC_CNTL( MICROMAG_DRDY_VIC_SLOT) = VIC_ENABLE | MS2100_DRDY_VIC_IT;
        _VIC_ADDR( MICROMAG_DRDY_VIC_SLOT) = (unsigned int) EXTINT_ISR; //
            address of the ISR


        package_read.bit_mode = SPI_8_BIT_MODE;
        /* trigger 2 bytes read */
        package_read.data[0] = 0;
        package_read.data[1] = 0;
        package_read.length = 2;
        package_read.slave_select = &ms2100_select;
        package_read.slave_unselect = &ms2100_unselect;
        package_read.spi_interrupt_handler = &ms2100_on_spi_int_read;

        package_req.bit_mode = SPI_8_BIT_MODE;
        package_req.length = 1;
        package_req.slave_select = &ms2100_select_and_reset;
        package_req.slave_unselect = &ms2100_unselect;
        package_req.spi_interrupt_handler = &ms2100_on_spi_int_req;
}


int ms2100_get_status(void){
        return ms2100_status;
}


void ms2100_send_request(uint8_t axis) {
        if(ms2100_status!=MS2100_IDLE){
                return;//if an other measurement is in process return
        }
        ms2100_status = MS2100_SENDING_REQ; //set status
        ms2100_current_axis = axis;
        ms2100_sampling_nr = 0;
        unsigned char cmd = 2 << 4 | (ms2100_current_axis+1) << 0;
        package_req.data[0] = cmd;
        spi_transmit(&package_req);
}


short ms2100_get_value(int axis){
        return (short)(0.0373*ms2100_values[axis][0] + 0.0639*ms2100_values[
            axis][1] + 0.1005*ms2100_values[axis][2] + 0.1326*ms2100_values[axis
            ][3] + 0.1514*ms2100_values[axis][4] + 0.1514*ms2100_values[axis
            ][5]+ 0.1326*ms2100_values[axis][6] + 0.1005*ms2100_values[axis][7]
            + 0.0639*ms2100_values[axis][8] + 0.0373*ms2100_values[axis][9]);
//      return (short)ms2100_values[axis][0];
}


static void ms2100_on_spi_int_read(void)
{
```

```
        //uint8_t buffer[200];  // string buffer for debug messages
        //sprintf((char *)buffer, "ms2100 read done\n");
        //message_send_debug(COMM_0, buffer);

        short new_val;
        new_val = SSPDR << 8;
        new_val += SSPDR;

        if((new_val) > 300 || new_val < -300){
                //there was a outliner we take the stay at the last measurement
        }
        else{
                ms2100_values[ms2100_current_axis][ms2100_sampling_nr] =
                    new_val;
        }

        ms2100_sampling_nr++;

        if(ms2100_sampling_nr>=MS2100_SAMPLING_TIMES){
                ms2100_status = MS2100_IDLE;
        }
        else{
                ms2100_status = MS2100_SENDING_REQ;
                spi_transmit(&package_req);
        }

        return;
}


static void ms2100_on_spi_int_req(void){
        /* read dummy control byte reply */
        //uint8_t buffer[200];  // string buffer for debug messages
        //sprintf((char *)buffer, "ms2100 request done\n");
        //message_send_debug(COMM_0, buffer);

        unsigned char foo __attribute__((unused)) = SSPDR;
        ms2100_status = MS2100_WAITING_EOC;

        return;
}

static void ms2100_do_before_spi_transmit(void) {

        MS2100_SS_IOCLR|=1<<MS2100_SS_PIN;

        uint32_t start_time_tc;
        start_time_tc = T0TC;

        //uint8_t buffer[200];  // string buffer for debug messages
        //sprintf((char *)buffer, "ms2100 before transmit 1\n");
        //message_send_debug(COMM_0, buffer);


        if (MAX_TC_VAL - start_time_tc >= 2*MS2100_RESET_TIMEOUT_BASE) {

                while(T0TC-start_time_tc < 2*MS2100_RESET_TIMEOUT_BASE) ;
```

```
        }
        else {

                while(T0TC<2*MS2100_RESET_TIMEOUT_BASE-1) ;

        }

        ms2100_set();
        if(MAX_TC_VAL - start_time_tc >= 3*MS2100_RESET_TIMEOUT_BASE) {
                while(T0TC-start_time_tc < 3*MS2100_RESET_TIMEOUT_BASE) ;
        }
        else {
                while(T0TC<3*MS2100_RESET_TIMEOUT_BASE-1) ;
        }
        ms2100_reset();
        //uint8_t buffer[200];  // string buffer for debug messages
        //sprintf((char *)buffer, "ms2100 before transmit 2\n");
        //message_send_debug(COMM_0, buffer);

}


static void EXTINT_ISR(void) {
        ISR_ENTRY();

        //uint8_t buffer[200];  // string buffer for debug messages
        //sprintf((char *)buffer, "ms2100 interrupt\n");
        //message_send_debug(COMM_0, buffer);

    ms2100_status = MS2100_READING_RES; //set status
        spi_transmit(&package_read);

        /* clear EINT */
        EXTINT|=  1 << MS2100_DRDY_EINT;

        VICVectAddr = 0x00000000; /* clear this interrupt from the VIC */
        ISR_EXIT();
}
```

C.11. On-Chip A/D Converter Driver

C.11.1. adc.h

```
#ifndef ADC_H_
#define ADC_H_

#include "inttypes.h"
#include "conf.h"

#if defined(BOARD_PIXHAWK_V100) || defined(BOARD_TWOG_BOOZ)
#define ADC_BAT_VDC_CHANNEL                    4
#define ADC_BPLANCD_IO_0_CHANNEL        0
#define ADC_BPLANCD_IO_2_CHANNEL        2
#endif



#if defined(IMU_PIXHAWK_V200) || defined(IMU_PIXHAWK_V210)
#define ADC_BAT_VDC_CHANNEL                    5
#define ADC_3_CHANNEL                          6
#define ADC_5_CHANNEL                          2
#define ADC_6_CHANNEL                          1
#define ADC_7_CHANNEL                          3
#endif

void adc_init(void);

uint16_t adc_get_value(uint8_t channel);

#endif /*ADC_H_*/
```

C.11.2. adc.c

```
#include "adc.h"
#include "conf.h"
#include "LPC21xx.h"


#if defined(IMU_PIXHAWK_V200) || defined(IMU_PIXHAWK_V210)
void adc_init(void){

        int CLKDIV = (int) (PCLK / 4500000);
        if(CLKDIV>0){
                CLKDIV=CLKDIV-1;
        }

        //set ADC3 (P0.4) as AD0.6
        PINSEL0 |= (3<<8);

        //set ADC5 (P0.29) as AD0.2
        PINSEL1 |= (3<<26);

        //set ADC6 (P0.28) as AD0.1
        PINSEL1 |= (3<<24);
```

```
        //set ADC7 (P0.12) as AD1.3
        PINSEL0 |= (3<<24);

        //set BAT_VDC (P0.15) as AD1.5
        PINSEL0 |= (3<<30);

        AD0CR = ( 1<<1 | 1<<2 | 1<<6 | 1<<16 | CLKDIV<<8 | 1<<21);
        AD1CR = ( 1<<3 | 1<<5 | 1<<16 | CLKDIV<<8 | 1<<21);
}

uint16_t adc_get_value(uint8_t channel){
        uint16_t adc_value=0;
        switch(channel){
        case ADC_BAT_VDC_CHANNEL:
                adc_value=(AD1DR5>>6)&0x03FF;
                break;
        case ADC_3_CHANNEL:
                adc_value=(AD0DR6>>6)&0x03FF;
                break;
        case ADC_5_CHANNEL:
                adc_value=(AD0DR2>>6)&0x03FF;
                break;
        case ADC_6_CHANNEL:
                adc_value=(AD0DR1>>6)&0x03FF;
                break;
        case ADC_7_CHANNEL:
                adc_value=(AD1DR3>>6)&0x03FF;
                break;
        }
        return adc_value;
}

#endif


#if defined(BOARD_PIXHAWK_V100) || defined(BOARD_TWOG_BOOZ)

void adc_init(void){

        int CLKDIV = (int) (PCLK / 4500000);
        if(CLKDIV>0){
                CLKDIV=CLKDIV-1;
        }

        //set BPlan_IO_0 (P0.6) as AD1.0
//      PINSEL0 |= (3<<12);

        //set BPlan_IO_2 (P0.10) as AD1.2
        PINSEL0 |= (3<<20);

        //set BAT_VDC (P0.13) as AD1.4
        PINSEL0 |= (3<<26);

        AD1CR = ( 1<<0 | 1<<2 | 1<<4 | 1<<16 | CLKDIV<<8 | 1<<21);
}
```

```
uint16_t adc_get_value(uint8_t channel){
        uint16_t adc_value=0;
        switch(channel){
        case ADC_BAT_VDC_CHANNEL:
                adc_value=(AD1DR4>>6)&0x03FF;
                break;
        case ADC_BPLANCD_IO_0_CHANNEL:
                adc_value=(AD1DR0>>6)&0x03FF;
                break;
        case ADC_BPLANCD_IO_2_CHANNEL:
                adc_value=(AD1DR2>>6)&0x03FF;
                break;
        }
        return adc_value;
}

#endif
```

C.12. Battery Charge Estimation (battery.h)

```
/*
 * battery.h
 *
 *  Created on: March 1, 2010
 *      Author: Christian Dobler
 */

#ifndef BATTERY_H_
#define BATTERY_H_

#include "inttypes.h"
#include "adc.h"
#include "conf.h"

#if defined(BOARD_PIXHAWK_V100) || defined(BOARD_TWOG_BOOZ)
/**
 * This function reads out the battery voltage
 * @return distance from the battery voltage in mv
 */
static inline uint16_t battery_get_value(void){
        uint16_t adc_value=adc_get_value(ADC_BAT_VDC_CHANNEL);
        uint16_t bat_act_mv = (uint16_t)(0.09121*adc_value*adc_value-81.95*
            adc_value+26480);
        return (bat_act_mv);
}

#endif

#if defined(IMU_PIXHAWK_V200) || defined(IMU_PIXHAWK_V210)
/**
 * This function reads out the battery voltage
 * @return distance from the battery voltage in mv
 */
static inline uint16_t battery_get_value(void){
        uint16_t adc_value=adc_get_value(ADC_BAT_VDC_CHANNEL);
        uint16_t bat_act_mv = (uint16_t) (BAT_VOLT_SCALE*(float)adc_value);
        return (bat_act_mv);
}

#endif

#endif /* BATTERY_H_ */
```

C.13. Configuration Files

C.13.1. conf.h

```
/*=====================================================================

PIXHAWK mavlib - The Micro Air Vehicle Platform Library
Please see our website at <http://pixhawk.ethz.ch>


Original Authors:
  Martin Rutschmann <mavteam@student.ethz.ch>
  Dominik Honegger <mavteam@student.ethz.ch>
Contributing Authors (in alphabetical order):
  Lorenz Meier <lm@student.ethz.ch>
  Christian Dobler <mavteam@student.ethz.ch>

Todo:


(c) 2009 PIXHAWK PROJECT

This file is part of the PIXHAWK project

    mavlib is free software: you can redistribute it and/or modify
    it under the terms of the GNU General Public License as published by
    the Free Software Foundation, either version 3 of the License, or
    (at your option) any later version.

    mavlib is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
    GNU General Public License for more details.

    You should have received a copy of the GNU General Public License
    along with mavlib. If not, see <http://www.gnu.org/licenses/>.

=====================================================================*/

/**
* @file conf.h
*
* @brief General Configuration
*
* These settings are general and valid across boards.
*
**/

#ifndef _CONF_H_
#define _CONF_H_

#include "stdbool.h"


// Select the board
// has to be BOARD_PIXHAWK_V100, BOARD_TWOG_BOOZ, IMU_PIXHAWK_V200 or
    IMU_PIXHAWK_V210
```

```
#define IMU_PIXHAWK_V200

//############### GENERAL SETUP ####################

#define ONBOARD_PARAM_COUNT 30            ///< Store 30 parameters in EEPROM and
    expose them over MAVLink paramter interface
#define ONBOARD_PARAM_NAME_LENGTH 15      ///< Parameter names are transmitted
    with max. 15 chars over MAVLink
#define ACID 42
#define SW_VERSION 2000
#define PPM_SAFETY_SWITCH_CHANNEL 5       ///< The PPM remote control channel
    where the safety switch is attached to
#define POSITION_ACCEPT_CONFIDENCE 0.8f ///< The position confidence which is
    considered safe
#define POSITION_ACCEPT_TIMEOUT 1000000 ///< The position is accepted as valid
    if not older than 1s (1 million usecs)

/*Downlink
    ******************************************************************************
    */

//#define DOWNLINK_USE_UART1
//#define DOWNLINK_BAUD                   115200
//#define DOWNLINK_MODE          UART_8N1

/* Comm settings ********************************************************/

//#define MAV_MAX_DIFFERENT_PACKAGES 256
#define COMM_UART_BAUD   57600
#define COMM_UART_MODE   UART_8N1

/* Periodic events ***************************************/

#define PERIODIC_TASK_SEC          20e-3
#define PERIODIC_TASK_PERIOD SYS_TICS_OF_SEC( PERIODIC_TASK_SEC )
#define START_TIME_PERIODIC SYS_TICS_OF_USEC( 1000000 )
#define START_TIME_PWM (START_TIME_PERIODIC + SYS_TICS_OF_USEC( 10000 ))

/* Interrupt Vector Slots, Priority: 1=highest
    ***************************************/

#define SSP_VIC_SLOT                      1
#define I2C_VIC_SLOT                      2
#define TIMER0_VIC_SLOT                   3
#define UART1_VIC_SLOT                    5
#define UART0_VIC_SLOT                    6
#define MICROMAG_DRDY_VIC_SLOT   7
#define PRESSURE_EOC_VIC_SLOT    8
#define MAX1168_EOC_VIC_SLOT     12
#define SCP1000_EOC_VIC_SLOT     13

/*! Number of channels which should be received by PPM IN */
#define PPM_NB_CHANNEL 9
/*! The minimum length of the pause of the ppm signal, used to detect failures
    */
#define PPM_SYNC_MIN_LEN 3000
```

```
/*! The maximum length of the pause of the ppm signal, used to detect failures
    */
#define PPM_SYNC_MAX_LEN 15000
/*! The minimum length of the data of the ppm signal, used to detect failures
    */
#define PPM_DATA_MIN_LEN 800
/*! The maximum length of the data of the ppm signal, used to detect failures
    */
#define PPM_DATA_MAX_LEN 2200

/* Comm settings ************************************************************/

/*  ****************************************************************/

#define WATCHDOG_NB_OF_CHANNELS 1

#define WATCHDOG_CHANNEL_PERIODIC 0
#define WATCHDOG_CHANNEL_PERIODIC_TIMEOUT SYS_TICS_OF_USEC(30000)

/*  ****************************************************************/

#define BAT_VOLT_SCALE            15.58


//static uint32_t watchdog_time_out_times[WATCHDOG_NB_OF_CHANNELS] = {
    WATCHDOG_CHANNEL_PERIODIC_TIMEOUT};

/*  ****************************************************************/


//################ INCLUDE BOARDS #####################

#ifdef BOARD_PIXHAWK_V100
#include "board_conf_pixhawk_v100.h"
#endif

#ifdef BOARD_TWOG_BOOZ
#include "board_conf_twog+booz.h"
#endif

#ifdef IMU_PIXHAWK_V200
#include "imu_conf_v200.h"
#endif

#ifdef IMU_PIXHAWK_V210
#include "imu_conf_v210.h"
#endif

#define PIXHAWK_2010_MZ
#ifdef PIXHAWK3
#include "pixhawk_3.h"
#endif

#ifdef PIXHAWK2
#include "pixhawk_2.h"
#endif
```

```
#ifdef PIXHAWK_2010_OZ
#include "pixhawk_2010_oz.h"
#endif

#ifdef PIXHAWK_2010_MZ
#include "pixhawk_2010_mz.h"
#endif


#endif /* _CONF_H_ */
```

C.13.2. imu_conf_v200.h

```
/*=====================================================================

PIXHAWK mavlib - The Micro Air Vehicle Platform Library
Please see our website at <http://pixhawk.ethz.ch>


Original Authors:
  Christian Dobler <mavteam@student.ethz.ch>
  Martin Rutschmann <mavteam@student.ethz.ch>
  Dominik Honegger <mavteam@student.ethz.ch>
  Lorenz Meier <lm@student.ethz.ch>
Contributing Authors (in alphabetical order):



Todo:


(c) 2009 PIXHAWK PROJECT

This file is part of the PIXHAWK project

    mavlib is free software: you can redistribute it and/or modify
    it under the terms of the GNU General Public License as published by
    the Free Software Foundation, either version 3 of the License, or
    (at your option) any later version.

    mavlib is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
    GNU General Public License for more details.

    You should have received a copy of the GNU General Public License
    along with mavlib. If not, see <http://www.gnu.org/licenses/>.

=====================================================================*/

/**
* @file imu_conf_v200.h
*
* @brief Settings for the initial revision of the PIXHAWK flight board
*
```

```
**/

#ifndef IMU_CONF_V200_H_
#define IMU_CONF_V200_H_

#define CPU_ARM7

/*! @brief Master oscillator freq.        */
#define FOSC (12000000)
/*! @brief PLL multiplier                 */
#define PLL_MUL (5)
/*! @brief CPU clock freq.                */
#define CCLK (FOSC * PLL_MUL)
/*! @brief Peripheral bus speed mask 0x00->4, 0x01-> 1, 0x02 -> 2    */
#define PBSD_BITS 0x00
/*! @brief Peripheral bus clock divider  */
#define PBSD_VAL 4
/*! @brief Peripheral bus clock freq.    */
#define PCLK (CCLK / PBSD_VAL)


/*==============================================================================*/


/** @name LED
 *  Pin settings for the LEDs. */
//@{
// The green led is on pin P1.17 (as defined by IO1 port and pin 17)

/*! LED pin. This is the pin on witch the led is connected. */
#define LED_GREEN                        17
/*! GPIO direction register for LED (IO0DIR or IO1DIR) */
#define LED_GREEN_DIR            IO1DIR
/*! GPIO clear register for LED (IO0CLR or IO1CLR) */
#define LED_GREEN_CLEAR          IO1CLR
/*! GPIO set register for LED (IO0SET or IO1SET) */
#define LED_GREEN_SET            IO1SET
/*! GPIO pin register for LED (IO0PIN or IO1PIN) */
#define LED_GREEN_PIN            IO1PIN

// The red led is on pin P1.16 (as defined by IO1 port and pin 16)
/*! LED pin. This is the pin on witch the led is connected. */
#define LED_RED                  16
/*! GPIO direction register for LED (IO0DIR or IO1DIR) */
#define LED_RED_DIR              IO1DIR
/*! GPIO clear register for LED (IO0CLR or IO1CLR) */
#define LED_RED_CLEAR    IO1CLR
/*! GPIO set register for LED (IO0SET or IO1SET) */
#define LED_RED_SET              IO1SET
/*! GPIO pin register for LED (IO0PIN or IO1PIN) */
#define LED_RED_PIN              IO1PIN
//@}


/*==============================================================================*/
```

```
/** @name PPM
 *  Settings for recording a ppm input from a remote control */
//@{
// (P0.4) is used as PPM in (CAP0.2)
/*! The settings for P0.6 are on Pinsel0 */
#define PPM_CAPTURE_PINSEL                                          PINSEL0
/*! Choose pin 2 (hence CAP0.2) */
#define PPM_CAPTURE_PINSEL_VAL                              2
/*! The 12 and the 13 bit of pinsel 0 are for the P0.6 settings */
#define PPM_CAPTURE_PINSEL_BIT                              12
/*! Timer interrupt number from capture register */
#define PPM_CAPTURE_TIMER_INTERRUPT                 TIR_CR2I
/*! Capture control register (T0CCR if ppm is on timer0, T1CCR if it is on
    timer1) */
#define PPM_CAPTURE_CONTROL_REGISTER                T0CCR
/*! enable capture 0.2 on falling edge + trigger interrupt */
#define PPM_CAPTURE_CONTROL_REGISTER_VALUE          TCCR_CR2_F | TCCR_CR2_I
/*! capture register with the captured value */
#define PPM_CAPTURE_CAPTURE_REGISTER                T0CR2
//@}

//Slave Select PIN//////////////////////////////////////////////////////
/* Micromag on SSP, IMU connector */
//Slave select pin on P1.22
#define MS2100_SS_PIN     22
#define MS2100_SS_IODIR  IO0DIR
#define MS2100_SS_IOSET  IO0SET
#define MS2100_SS_IOCLR  IO0CLR

//Magnet reset pin//////////////////////////////////////////////////////
//Magnet reset pin on P0.25
#define MS2100_RESET_PIN     25
#define MS2100_RESET_IODIR  IO0DIR
#define MS2100_RESET_IOSET  IO0SET
#define MS2100_RESET_IOCLR  IO0CLR

//DATAREADY PIN/////////////////////////////////////////////////////////
//Dataready pin on P0.30/EINT3
#define MS2100_DRDY_PINSEL  PINSEL1
#define MS2100_DRDY_PINSEL_BIT  28
#define MS2100_DRDY_PINSEL_VAL  2
#define MS2100_DRDY_EINT  3
#define MS2100_DRDY_VIC_IT  VIC_EINT3

/*==============================================================================*/

/** @name SCA3100
 *  Settings for sca3100 sensor */
//@{
#define SCA3100_SS_PIN     18
#define SCA3100_SS_IODIR  IO1DIR
#define SCA3100_SS_IOSET  IO1SET
#define SCA3100_SS_IOCLR  IO1CLR

#define SCA3100_X_AXIS  0
#define SCA3100_Y_AXIS  1
#define SCA3100_Z_AXIS  2
```

```
//@}

/*==========================================================================*/

/** @name ACCELERATION
 *  Settings for acceleration sensors */
//@{
/*! Neutral output of the adc for the x acceleration */
#define ACC_X_NEUTRAL 32835
/*! Neutral output of the adc for the y acceleration */
#define ACC_Y_NEUTRAL 32760
/*! Neutral output of the adc for the z acceleration */
#define ACC_Z_NEUTRAL 33144
//@}


/*==========================================================================*/

/** @name MAGNETOMETER
 *  Settings for magnetometer sensors */
//@{
#define MAG_SENSOR_OFFSET_X      -4
#define MAG_SENSOR_OFFSET_Y      -7
#define MAG_SENSOR_OFFSET_Z      11
#define MAG_MOTOR_OFFSET_X       156
#define MAG_MOTOR_OFFSET_Y       5
#define MAG_MOTOR_OFFSET_Z       -202

#define MAG_RANGE_FACTOR_X (int)(1000*1)
#define MAG_RANGE_FACTOR_Y (int)(1000*73/67)
#define MAG_RANGE_FACTOR_Z (int)(1000*64/112)
//@}

/**
 * @name ANALOG TO DIGITAL CONVERTER (ADC)
 */
//@{
// XY-GYRO (y = roll, x = pitch) ADC
#define ADS8341_0                        0
#define ADS8341_0_SS_IODIR       IO1DIR
#define ADS8341_0_SS_IOSET       IO1SET
#define ADS8341_0_SS_IOCLR       IO1CLR
#define ADS8341_0_SS_IOPIN       IO1PIN
// P1.25 (as defined by using IO1DIR/IO1SET and number 25)
// named CS XY-GYRO PRESSURE in schematic
#define ADS8341_0_SS_PIN         25

//@}
/*==========================================================================*/


/** @name GYROS
 *  Settings for gyros sensors */
//@{
/*! Neutral output of the adc for the yaw gyro */
#define GYROS_YAW_NEUTRAL        32349
/*! Neutral output of the adc for the pitch gyro */
```

```
#define GYROS_PITCH_NEUTRAL        34253
/*! Neutral output of the adc for the roll gyro */
#define GYROS_ROLL_NEUTRAL         32107

/*! Channel of the max1168 analog input of the roll gyro sensor */
#define GYROS_ROLL_ADS8341_0_CHANNEL      1
/*! Channel of the max1168 analog input of the pitch gyro sensor */
#define GYROS_PITCH_ADS8341_0_CHANNEL     0
/*! Channel of the max1168 analog input of the yaw gyro sensor */
#define GYROS_YAW_ADS8341_0_CHANNEL                2

//@}

/*==============================================================================*/


/**
 * @name TEMPERATURE
 *
 * Setting for the temperature readout.
 */
//@{
/*! Digital Analog Converter settings. The DAC pin is fixed to P0.25/AD0.4/AOUT
    . */
#define TEMPERATURE_ADS8341_0_CHANNEL     3



//@}

/*==============================================================================*/


/** @name BAROMETER
 *  Settings for barometer readout, including DAC. The Digital Analog Converter
      is used
 *  to adjust dynamically the offset of the barometer, allowing a very high
     precision. */
//@{
/*! Digital Analog Converter settings. The DAC pin is fixed to P0.25/AD0.4/AOUT
    . */
/*#define DAC_VALUE_MASK          (0x3FF)
#define DAC_PINSEL                PINSEL1
#define DAC_PINSEL_VAL   2
#define DAC_PINSEL_BIT   18

#define PRESSURE_ADS8341_0_CHANNEL 3
*/
//@}


/*==============================================================================*/


/** @name MOTORS
 *  Settings for motors */
//@{
```

```
/*! The pwm channel of lower motor  */
//#define MOTORS_LOWER_PWM_CHANNEL 3
/*! The pwm channel of upper motor  */
//#define MOTORS_UPPER_PWM_CHANNEL 2
//@}


/*==========================================================================*/


/** @name SERVOS
 *  Settings for servos */
//@{
/*! The pwm channel of servo 1 */
#define SERVOS_SERVO_NEGY_PWM_CHANNEL 5
/*! The pwm channel of servo 2 */
#define SERVOS_SERVO_POSY_PWM_CHANNEL 0
//@}


/*==========================================================================*/



/*==========================================================================*/


/** @name UART
 *  Settings for the two UART ports of the arm7 */
//@{
/*! Uart 0 receive buffer size (in Bytes) */
#define UART0_RX_BUFFER_SIZE 512
/*! Uart 0 transmit buffer size (in Bytes) */
#define UART0_TX_BUFFER_SIZE 512

/*! Uart 1 receive buffer size (in Bytes) */
#define UART1_RX_BUFFER_SIZE 512
/*! Uart 1 transmit buffer size (in Bytes) */
#define UART1_TX_BUFFER_SIZE 512
//@}


/*==========================================================================*/


/** @name PWM
 *  Settings for the pwm generation with the 4017 decade counter */
//@{
/* P0.5 as MAT0.1  */
#define PWM_CLOCK_PIN  5
#define PWM_CLOCK_PINSEL PINSEL0
#define PWM_CLOCK_PINSEL_VAL 0x02
#define PWM_CLOCK_PINSEL_BIT 10
/* p0.20 as GPIO */
#define PWM_RESET_PIN 20
#define PWM_RESET_IODIR IO1DIR
#define PWM_RESET_IOSET IO1SET
```

```
#define PWM_RESET_IOCLR IO1CLR
/* Number of used channels */
#define PWM_NB_CHANNELS 9
/* minumum pulse width in usec */
#define PWM_MIN_PULSE_USEC 1000
/* maximum pulse width in usec */
#define PWM_MAX_PULSE_USEC 2000
/* length of the pwm periode in usec */
#define PWM_PERIODE SYS_TICS_OF_USEC(20000)
//@}


/*==============================================================================*/


/** @name CAM_TRIGGER
 *  Settings for the cam trigger signal */
//@{
/* P0.21 (IRV on TWOG) as PWM5 output */
//#define CAM_TRIGGER_PIN_PINSEL          PINSEL1
//#define CAM_TRIGGER_PIN_VAL             0x01
//#define CAM_TRIGGER_PIN_BIT             10

/* We use a PWM5 output, so the matchregister has to be PWMMR5 */
//#define CAM_TRIGGER_PWM_CHANNEL_MR                    PWMMR5
/* We use a PWM5 output, this has to be enabled by writing PWMPCR_ENA5 into
   PWMPCR */
//#define CAM_TRIGGER_PWM_CHANNEL_PCR_ENABLE     PWMPCR_ENA5

/* Period of the cam trigger pulse signal in us */
//#define CAM_TRIGGER_PERIODE_US                 100000
/* Pulse width of the cam trigger pulse signal in us */
//#define CAM_TRIGGER_PULSE_WIDTH_US     10000
//@}


/*==============================================================================*/
/*============Following conf not commented yet==================================*/
/*==============================================================================*/



/* SPI settings ***********************************************************/

/* SSPCR0 settings */
#define SSP_DDS   0x0F << 0  /* data size          : 16 bits        */
#define SSP_FRF   0x00 << 4  /* frame format       : SPI            */
#define SSP_CPOL  0x00 << 6  /* clock polarity     : data captured on first clock
     transition */
#define SSP_CPHA  0x00 << 7  /* clock phase        : SCK idles low */
#define SSP_SCR   0x0F << 8  /* serial clock rate : divide PCLK by 16*SSPCPSR */

/* SSPCR1 settings */
#define SSP_LBM   0x00 << 0  /* loopback mode      : disabled                   */
#define SSP_SSE   0x00 << 1  /* SSP enable         : disabled                   */
#define SSP_MS    0x00 << 2  /* master slave mode : master                      */
```

```
#define SSP_SOD   0x00 << 3  /* slave output disable : don't care when master */

#define SSPCR0_VAL (SSP_DDS |  SSP_FRF | SSP_CPOL | SSP_CPHA | SSP_SCR )
#define SSPCR1_VAL (SSP_LBM |  SSP_SSE | SSP_MS | SSP_SOD )

#define SSP_PINSEL1_SCK   (2<<2)
#define SSP_PINSEL1_MISO  (2<<4)
#define SSP_PINSEL1_MOSI  (2<<6)

#define SPI_PACKAGE_BUFFER_SIZE 10

/*  ******************************************************************/

/* I2C settings ********************************************************/

#define I2C_WRITE                                        0
#define I2C_READ                                         1

#define I2C_PACKAGE_BUFFER_SIZE          10
#define MAX_I2C_PACKAGE_SIZE             32
#define LPC_I2C_ADR                                      0x20
#define I2C_PERMANENT_ERROR_LIMIT        1000

/* I2C0 settings */
/* I2C0 clock settings: I2C_CLOCK = PCLK/(I2C0SCLH_VAL+I2C0SCLL_VAL) = 1MHz */
#define I2C0SCLH_VAL             7                /* number of PCLK clock cycles
    that SCL stays high */
#define I2C0SCLL_VAL             8                /* number of PCLK clock cycles
    that SCL stays high */
/* I2C0 mode setting */
#define I2C0CONSET_VAL           0x40     /* master mode only */

#define I2C0_PINSEL0_SCL         (1<<4)
#define I2C0_PINSEL0_SDA         (1<<6)


/* I2C1 settings */
/* I2C1 clock settings: I2C_CLOCK = PCLK/(I2C0SCLH_VAL+I2C0SCLL_VAL) = 1MHz */
#define I2C1SCLH_VAL             7                /* number of PCLK clock cycles
    that SCL stays high */
#define I2C1SCLL_VAL             8                /* number of PCLK clock cycles
    that SCL stays high */
/* I2C1 mode setting */
#define I2C1CONSET_VAL           0x40     /* master mode only */

#define I2C1_PINSEL0_SCL         (3<<22)
#define I2C1_PINSEL0_SDA         (3<<28)

/*  ******************************************************************/

/*********************************************************/

/** @name PRESSURE **/

#define BMP085_I2C_BUS_NUMBER                            1

#define BMP085_PRESSURE_CONVERSION_MODE          0    //0, 1 or 2
```

```
/*  ******************************************************************/

/*  ******************************************************************/

/*********************************************************/

/** @name EEPROM **/

#define EEPROM_I2C_BUS_NUMBER                                   1

#define EEPROM_I2C_SLAVE_ADDRESS                                0xA0    ///< I2C slave
    address of Microchip 24FC256 EEPROM (0b10100000)


/*  ******************************************************************/

/*********************************************************/

/** @name I2C Motor Controllers **/

#define MOT_I2C_BUS_NUMBER                                              0

#define MOT1_I2C_SLAVE_ADDRESS                          0xB2    ///< I2C slave address
    of I2C motor controller number 1
#define MOT2_I2C_SLAVE_ADDRESS                          0xB4    ///< I2C slave address
    of I2C motor controller number 2
#define MOT3_I2C_SLAVE_ADDRESS                          0xB6    ///< I2C slave address
    of I2C motor controller number 3
#define MOT4_I2C_SLAVE_ADDRESS                          0xB8    ///< I2C slave address
    of I2C motor controller number 4


/*  ******************************************************************/

#endif /* IMU_CONF_V200_H_ */
```

C.13.3. imu_conf_v210.h

```
/*=====================================================================

PIXHAWK mavlib - The Micro Air Vehicle Platform Library
Please see our website at <http://pixhawk.ethz.ch>


Original Authors:
  Christian Dobler <mavteam@student.ethz.ch>
  Martin Rutschmann <mavteam@student.ethz.ch>
  Dominik Honegger <mavteam@student.ethz.ch>
  Lorenz Meier <lm@student.ethz.ch>
Contributing Authors (in alphabetical order):



Todo:
```

```
(c) 2009 PIXHAWK PROJECT

This file is part of the PIXHAWK project

    mavlib is free software: you can redistribute it and/or modify
    it under the terms of the GNU General Public License as published by
    the Free Software Foundation, either version 3 of the License, or
    (at your option) any later version.

    mavlib is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
    GNU General Public License for more details.

    You should have received a copy of the GNU General Public License
    along with mavlib. If not, see <http://www.gnu.org/licenses/>.

=================================================================================*/

/**
* @file imu_conf_v200.h
*
* @brief Settings for the initial revision of the PIXHAWK flight board
*
**/

#ifndef IMU_CONF_V210_H_
#define IMU_CONF_V210_H_

#define CPU_ARM7

/*! @brief Master oscillator freq.        */
#define FOSC (12000000)
/*! @brief PLL multiplier                 */
#define PLL_MUL (5)
/*! @brief CPU clock freq.                */
#define CCLK (FOSC * PLL_MUL)
/*! @brief Peripheral bus speed mask 0x00->4, 0x01-> 1, 0x02 -> 2    */
#define PBSD_BITS 0x00
/*! @brief Peripheral bus clock divider  */
#define PBSD_VAL 4
/*! @brief Peripheral bus clock freq.    */
#define PCLK (CCLK / PBSD_VAL)


/*=================================================================================*/


/** @name LED
 *  Pin settings for the LEDs. */
//@{
// The green led is on pin P1.17 (as defined by IO1 port and pin 17)

/*! LED pin. This is the pin on witch the led is connected. */
#define LED_GREEN                          17
```

```
/*! GPIO direction register for LED (IO0DIR or IO1DIR) */
#define LED_GREEN_DIR            IO1DIR
/*! GPIO clear register for LED (IO0CLR or IO1CLR) */
#define LED_GREEN_CLEAR          IO1CLR
/*! GPIO set register for LED (IO0SET or IO1SET) */
#define LED_GREEN_SET            IO1SET
/*! GPIO pin register for LED (IO0PIN or IO1PIN) */
#define LED_GREEN_PIN            IO1PIN

// The red led is on pin P1.16 (as defined by IO1 port and pin 16)
/*! LED pin. This is the pin on witch the led is connected. */
#define LED_RED                  16
/*! GPIO direction register for LED (IO0DIR or IO1DIR) */
#define LED_RED_DIR              IO1DIR
/*! GPIO clear register for LED (IO0CLR or IO1CLR) */
#define LED_RED_CLEAR    IO1CLR
/*! GPIO set register for LED (IO0SET or IO1SET) */
#define LED_RED_SET              IO1SET
/*! GPIO pin register for LED (IO0PIN or IO1PIN) */
#define LED_RED_PIN              IO1PIN
//@}


/*==========================================================================*/


/** @name PPM
 *  Settings for recording a ppm input from a remote control */
//@{
// (P0.4) is used as PPM in (CAP0.2)
/*! The settings for P0.6 are on Pinsel0 */
#define PPM_CAPTURE_PINSEL                                          PINSEL0
/*! Choose pin 2 (hence CAP0.2) */
#define PPM_CAPTURE_PINSEL_VAL                              2
/*! The 12 and the 13 bit of pinsel 0 are for the P0.6 settings */
#define PPM_CAPTURE_PINSEL_BIT                              12
/*! Timer interrupt number from capture register */
#define PPM_CAPTURE_TIMER_INTERRUPT              TIR_CR1I
/*! Capture control register (T0CCR if ppm is on timer0, T1CCR if it is on
    timer1) */
#define PPM_CAPTURE_CONTROL_REGISTER             T0CCR
/*! enable capture 0.2 on falling edge + trigger interrupt */
#define PPM_CAPTURE_CONTROL_REGISTER_VALUE       TCCR_CR2_F | TCCR_CR2_I
/*! capture register with the captured value */
#define PPM_CAPTURE_CAPTURE_REGISTER             T0CR2
//@}

//Slave Select PIN//////////////////////////////////////////////////////
/* Micromag on SSP, IMU connector */
//Slave select pin on P1.22
#define MS2100_SS_PIN     22
#define MS2100_SS_IODIR  IO0DIR
#define MS2100_SS_IOSET  IO0SET
#define MS2100_SS_IOCLR  IO0CLR

//Magnet reset pin//////////////////////////////////////////////////////
//Magnet reset pin on P0.25
```

```
#define MS2100_RESET_PIN     25
#define MS2100_RESET_IODIR  IO0DIR
#define MS2100_RESET_IOSET  IO0SET
#define MS2100_RESET_IOCLR  IO0CLR

//DATAREADY PIN//////////////////////////////////////////////////////////
//Dataready pin on P0.30/EINT3
#define MS2100_DRDY_PINSEL  PINSEL1
#define MS2100_DRDY_PINSEL_BIT  28
#define MS2100_DRDY_PINSEL_VAL  2
#define MS2100_DRDY_EINT  3
#define MS2100_DRDY_VIC_IT  VIC_EINT3

/*==============================================================================*/

/** @name SCA3100
 *  Settings for sca3100 sensor */
//@{
#define SCA3100_SS_PIN     18
#define SCA3100_SS_IODIR  IO1DIR
#define SCA3100_SS_IOSET  IO1SET
#define SCA3100_SS_IOCLR  IO1CLR

#define SCA3100_X_AXIS  0
#define SCA3100_Y_AXIS  1
#define SCA3100_Z_AXIS  2
//@}

/*==============================================================================*/

/** @name ACCELERATION
 *  Settings for acceleration sensors */
//@{
/*! Neutral output of the adc for the x acceleration */
#define ACC_X_NEUTRAL  32835
/*! Neutral output of the adc for the y acceleration */
#define ACC_Y_NEUTRAL  32760
/*! Neutral output of the adc for the z acceleration */
#define ACC_Z_NEUTRAL  33144
//@}


/*==============================================================================*/

/** @name MAGNETOMETER
 *  Settings for magnetometer sensors */
//@{
#define MAG_SENSOR_OFFSET_X       -4
#define MAG_SENSOR_OFFSET_Y       -7
#define MAG_SENSOR_OFFSET_Z       11
#define MAG_MOTOR_OFFSET_X        156
#define MAG_MOTOR_OFFSET_Y        5
#define MAG_MOTOR_OFFSET_Z        -202

#define MAG_RANGE_FACTOR_X  (int)(1000*1)
#define MAG_RANGE_FACTOR_Y  (int)(1000*73/67)
#define MAG_RANGE_FACTOR_Z  (int)(1000*64/112)
```

```
//@}

/**
 * @name ANALOG TO DIGITAL CONVERTER (ADC)
 */
//@{
// XY-GYRO (y = roll, x = pitch) ADC
#define ADS8341_0                          0
#define ADS8341_0_SS_IODIR        IO1DIR
#define ADS8341_0_SS_IOSET        IO1SET
#define ADS8341_0_SS_IOCLR        IO1CLR
#define ADS8341_0_SS_IOPIN        IO1PIN
// P1.25 (as defined by using IO1DIR/IO1SET and number 25)
// named CS XY-GYRO PRESSURE in schematic
#define ADS8341_0_SS_PIN          25

//@}
/*==========================================================================*/


/** @name GYROS
 *  Settings for gyros sensors */
//@{
/*! Neutral output of the adc for the yaw gyro */
#define GYROS_YAW_NEUTRAL         32349
/*! Neutral output of the adc for the pitch gyro */
#define GYROS_PITCH_NEUTRAL       34253
/*! Neutral output of the adc for the roll gyro */
#define GYROS_ROLL_NEUTRAL        32107

/*! Channel of the max1168 analog input of the roll gyro sensor */
#define GYROS_ROLL_ADS8341_0_CHANNEL      1
/*! Channel of the max1168 analog input of the pitch gyro sensor */
#define GYROS_PITCH_ADS8341_0_CHANNEL     0
/*! Channel of the max1168 analog input of the yaw gyro sensor */
#define GYROS_YAW_ADS8341_0_CHANNEL                2

//@}

/*==========================================================================*/


/**
 * @name TEMPERATURE
 *
 * Setting for the temperature readout.
 */
//@{
/*! Digital Analog Converter settings. The DAC pin is fixed to P0.25/AD0.4/AOUT
    . */
#define TEMPERATURE_ADS8341_0_CHANNEL 3



//@}

/*==========================================================================*/
```

```
/** @name BAROMETER
 *  Settings for barometer readout, including DAC. The Digital Analog Converter
      is used
 *  to adjust dynamically the offset of the barometer, allowing a very high
     precision. */
//@{
/*! Digital Analog Converter settings. The DAC pin is fixed to P0.25/AD0.4/AOUT
    . */
/*#define DAC_VALUE_MASK          (0x3FF)
#define DAC_PINSEL                PINSEL1
#define DAC_PINSEL_VAL   2
#define DAC_PINSEL_BIT   18

#define PRESSURE_ADS8341_0_CHANNEL 3
*/
//@}


/*==============================================================================*/


/** @name MOTORS
 *  Settings for motors */
//@{
/*! The pwm channel of lower motor  */
//#define MOTORS_LOWER_PWM_CHANNEL 3
/*! The pwm channel of upper motor  */
//#define MOTORS_UPPER_PWM_CHANNEL 2
//@}


/*==============================================================================*/


/** @name SERVOS
 *  Settings for servos */
//@{
/*! The pwm channel of servo 1 */
#define SERVOS_SERVO_NEGY_PWM_CHANNEL 5
/*! The pwm channel of servo 2 */
#define SERVOS_SERVO_POSY_PWM_CHANNEL 0
//@}


/*==============================================================================*/


/*==============================================================================*/


/** @name UART
 *  Settings for the two UART ports of the arm7 */
//@{
/*! Uart 0 receive buffer size (in Bytes) */
#define UART0_RX_BUFFER_SIZE 512
```

```
/*! Uart 0 transmit buffer size (in Bytes) */
#define UART0_TX_BUFFER_SIZE 512

/*! Uart 1 receive buffer size (in Bytes) */
#define UART1_RX_BUFFER_SIZE 512
/*! Uart 1 transmit buffer size (in Bytes) */
#define UART1_TX_BUFFER_SIZE 512
//@}


/*==============================================================================*/


/** @name PWM
 *  Settings for the pwm generation with the 4017 decade counter */
//@{
/* P0.5 as MAT0.1  */
#define PWM_CLOCK_PIN  5
#define PWM_CLOCK_PINSEL PINSEL0
#define PWM_CLOCK_PINSEL_VAL 0x02
#define PWM_CLOCK_PINSEL_BIT 10
/* p0.20 as GPIO */
#define PWM_RESET_PIN 20
#define PWM_RESET_IODIR IO1DIR
#define PWM_RESET_IOSET IO1SET
#define PWM_RESET_IOCLR IO1CLR
/* Number of used channels */
#define PWM_NB_CHANNELS 9
/* minumum pulse width in usec */
#define PWM_MIN_PULSE_USEC 1000
/* maximum pulse width in usec */
#define PWM_MAX_PULSE_USEC 2000
/* length of the pwm periode in usec */
#define PWM_PERIODE SYS_TICS_OF_USEC(20000)
//@}


/*==============================================================================*/


/** @name CAM_TRIGGER
 *  Settings for the cam trigger signal */
//@{
/* P0.21 (IRV on TWOG) as PWM5 output */
//#define CAM_TRIGGER_PIN_PINSEL          PINSEL1
//#define CAM_TRIGGER_PIN_VAL             0x01
//#define CAM_TRIGGER_PIN_BIT             10

/* We use a PWM5 output, so the matchregister has to be PWMMR5 */
//#define CAM_TRIGGER_PWM_CHANNEL_MR                    PWMMR5
/* We use a PWM5 output, this has to be enabled by writing PWMPCR_ENA5 into
   PWMPCR */
//#define CAM_TRIGGER_PWM_CHANNEL_PCR_ENABLE     PWMPCR_ENA5

/* Period of the cam trigger pulse signal in us */
//#define CAM_TRIGGER_PERIODE_US                 100000
/* Pulse width of the cam trigger pulse signal in us */
```

```
//#define CAM_TRIGGER_PULSE_WIDTH_US      10000
//@}


/*==========================================================================*/
/*==========Following conf not commented yet==============================*/
/*==========================================================================*/



/* SPI settings ***********************************************************/

/* SSPCR0 settings */
#define SSP_DDS   0x0F << 0  /* data size          : 16 bits        */
#define SSP_FRF   0x00 << 4  /* frame format       : SPI            */
#define SSP_CPOL  0x00 << 6  /* clock polarity     : data captured on first clock
     transition */
#define SSP_CPHA  0x00 << 7  /* clock phase        : SCK idles low */
#define SSP_SCR   0x0F << 8  /* serial clock rate : divide PCLK by 16*SSPCPSR */

/* SSPCR1 settings */
#define SSP_LBM   0x00 << 0  /* loopback mode      : disabled                  */
#define SSP_SSE   0x00 << 1  /* SSP enable         : disabled                  */
#define SSP_MS    0x00 << 2  /* master slave mode : master                     */
#define SSP_SOD   0x00 << 3  /* slave output disable : don't care when master */

#define SSPCR0_VAL (SSP_DDS |  SSP_FRF | SSP_CPOL | SSP_CPHA | SSP_SCR )
#define SSPCR1_VAL (SSP_LBM |  SSP_SSE | SSP_MS | SSP_SOD )

#define SSP_PINSEL1_SCK   (2<<2)
#define SSP_PINSEL1_MISO  (2<<4)
#define SSP_PINSEL1_MOSI  (2<<6)

#define SPI_PACKAGE_BUFFER_SIZE 10

/*  ***********************************************************************/

/* I2C settings ***********************************************************/

#define I2C_WRITE                                        0
#define I2C_READ                                         1

#define I2C_PACKAGE_BUFFER_SIZE          10
#define MAX_I2C_PACKAGE_SIZE             100
#define LPC_I2C_ADR                                      0x20
#define I2C_PERMANENT_ERROR_LIMIT        1000

/* I2C0 settings */
/* I2C0 clock settings: I2C_CLOCK = PCLK/(I2C0SCLH_VAL+I2C0SCLL_VAL) = 1MHz */
#define I2C0SCLH_VAL             7                /* number of PCLK clock cycles
    that SCL stays high */
#define I2C0SCLL_VAL             8                /* number of PCLK clock cycles
    that SCL stays high */
/* I2C0 mode setting */
#define I2C0CONSET_VAL           0x40     /* master mode only */
```

```
#define I2C0_PINSEL0_SCL          (1<<4)
#define I2C0_PINSEL0_SDA          (1<<6)


/* I2C1 settings */
/* I2C1 clock settings: I2C_CLOCK = PCLK/(I2C0SCLH_VAL+I2C0SCLL_VAL) = 1MHz */
#define I2C1SCLH_VAL              7                /* number of PCLK clock cycles
    that SCL stays high */
#define I2C1SCLL_VAL              8                /* number of PCLK clock cycles
    that SCL stays high */
/* I2C1 mode setting */
#define I2C1CONSET_VAL            0x40     /* master mode only */

#define I2C1_PINSEL0_SCL          (3<<22)
#define I2C1_PINSEL0_SDA          (3<<28)

/*  ******************************************************************/

/*********************************************************/

/** @name PRESSURE **/

#define BMP085_I2C_BUS_NUMBER                                   1

#define BMP085_PRESSURE_CONVERSION_MODE           0    //0, 1 or 2

#define BMP085_EOC_PINSEL                                               PINSEL0
#define BMP085_EOC_PINSEL_VAL                                   3
#define BMP085_EOC_PINSEL_BIT                                   14
#define BMP085_EOC_EINT                                                 2

/*  ******************************************************************/

/*  ******************************************************************/

/*********************************************************/

/** @name EEPROM **/

#define EEPROM_I2C_BUS_NUMBER                                   1

#define EEPROM_I2C_SLAVE_ADDRESS                                0xA0     ///< I2C slave
    address of Microchip 24FC256 EEPROM (0b10100000)


/*  ******************************************************************/

/*********************************************************/

/** @name I2C Motor Controllers **/

#define MOT_I2C_BUS_NUMBER                                              0

#define MOT1_I2C_SLAVE_ADDRESS                    0xB2     ///< I2C slave address
    of I2C motor controller number 1
#define MOT2_I2C_SLAVE_ADDRESS                    0xB4     ///< I2C slave address
    of I2C motor controller number 2
```

```
#define MOT3_I2C_SLAVE_ADDRESS                    0xB6     ///< I2C slave address
    of I2C motor controller number 3
#define MOT4_I2C_SLAVE_ADDRESS                    0xB8     ///< I2C slave address
    of I2C motor controller number 4


/*  ****************************************************************/

#endif /* IMU_CONF_V210_H_ */
```

C.13.4. pixhawk_2010_mz.h

```
/*
 * pixhawk_2010_mz.h
 *
 *  Created on: Feb 22, 2010
 *      Author: Christian Dobler
 */

#ifndef PIXHAWK_2010_MZ_H_
#define PIXHAWK_2010_MZ_H_

//values for PH 2 Board
    ***********************************************************************/

#define ACCEL_NEUTRALX 2
#define ACCEL_NEUTRALY -38
#define ACCEL_NEUTRALZ 4
#define GYRO_NEUTRALX 29690
#define GYRO_NEUTRALY 30200
#define GYRO_NEUTRALZ 29860
//#define GYRO_NEUTRALZ 29439


#define MAG_NEUTRALX -56
#define MAG_NEUTRALY -17
#define MAG_SCALEX 0.0208333333f
#define MAG_SCALEY 0.0196078431f
#define DIP_ANGLE 1.0472f

#define GYRO_SCALE_X 0.001007 //unit[(rad/sec)/Gyro_Step] FROM MEASURED DATA
#define GYRO_SCALE_Y 0.000990
#define GYRO_SCALE_Z 0.001002

//PIXHAWK2
//#define ROLL_TRIMM -0.15f  //trimm value in [rad]
//#define NICK_TRIMM -0.167f //trimm value in [rad]

#endif /* PIXHAWK_2010_MZ_H_ */
```

C.13.5. pixhawk_2010_oz.h

```
/*
```

```
 * pixhawk_2010_oz.h
 *
 *  Created on: Feb 22, 2010
 *      Author: Christian Dobler
 */

#ifndef PIXHAWK_2010_OZ_H_
#define PIXHAWK_2010_OZ_H_

//values for PH 2 Board
    ***************************************************************************/

#define ACCEL_NEUTRALX 3
#define ACCEL_NEUTRALY 11
#define ACCEL_NEUTRALZ 22
#define GYRO_NEUTRALX 29683
#define GYRO_NEUTRALY 30232
#define GYRO_NEUTRALZ 29809
//#define GYRO_NEUTRALZ 29439


#define MAG_NEUTRALX -56
#define MAG_NEUTRALY -17
#define MAG_SCALEX 0.0208333333f
#define MAG_SCALEY 0.0196078431f
#define DIP_ANGLE 1.0472f

#define GYRO_SCALE_X 0.000955 //unit[(rad/sec)/Gyro_Step] FROM MEASURED DATA
#define GYRO_SCALE_Y 0.000823
#define GYRO_SCALE_Z 0.001010

//PIXHAWK2
//#define ROLL_TRIMM -0.15f  //trimm value in [rad]
//#define NICK_TRIMM -0.167f //trimm value in [rad]


#endif /* PIXHAWK_2010_OZ_H_ */
```