Hochschule Esslingen
University of Applied Sciences
Fakultät Informationstechnik

[HElikopter

Masterquad 2015

A Raspberry Pi based quadrocopter platform

A project work of the masters program
Software-based Automotive Systems (ASM-SB)
at Esslingen Graduate School

Oliver Breuning
Jürgen Schmidt
Martin Brodbeck
Phillip Woditsch

Period: Summer term 2015
Supervisor:
Prof. Dr. Jörg Friedrich
M Sc. Vikas Agrawal

Contents

List of Figures

List of Tables

Listings

Revision History

Revision	Date	Author(s)	Description
1.0	August 1, 2015	Juergen Schmidt, Oliver Breuning	created
2.0	October 15, 2015	Vikas Agrawal	Changes done for native Ubuntu machine
2.0	December 4, 2015	Vikas Agrawal	Rearranged the doucment for better readability

1 Project's context and goals

The very first target was to adapt a new hardware (Raspberry Pi) including a autonomous landing to the Quadrocopter from Hochschule Esslingen (HElikopter). As this was not enough work for all of us, the project was extended to include a autonomous landing, too. We were a team of 4 students who were interested in a project work with the HElikopter. We internally agreed in splitting the work in 2 parts. The other two students were more interested in developing the new hardware platform and software and we liked to care about the autonomous landing.

As we developed the autonomous landing to the new hardware, there was a slight dependence to each other. Following are the goals:

- Autonomous Landing
- Effective Distance measurement to ground
- Implementation on Raspberry Pi

This document describes a project in the context of the curriculum's module 'project work' of the masters program Software-based Automotive Systems. The project intends to set up a generic hardware and software platform to control the HElikopter Quadrocopters of Hochschule Esslingen, Faculty Information Technology. The platform shall be based on a real-time capable Linux Operating System and a Raspberry Pi B+ Board. Furthermore, all components used shall be available on market as much ready-made as possible.

Originally, the project was designed to be elaborated by a group of four students. Unfortunately, two students had to face some substantial problems during the project work. Eventually, the project got split up into two separated groups. In consequence, the initially intended goals could not be reached completely. This is partly reflected in the below shown tasks, milestones and project goals.

Nevertheless, the authors of this document are confident that the shown Raspberry Pi based platform is a solid and stable approach. With some extensions on this work, a Linux-driven Quadrocopter can be realized in a future project work.

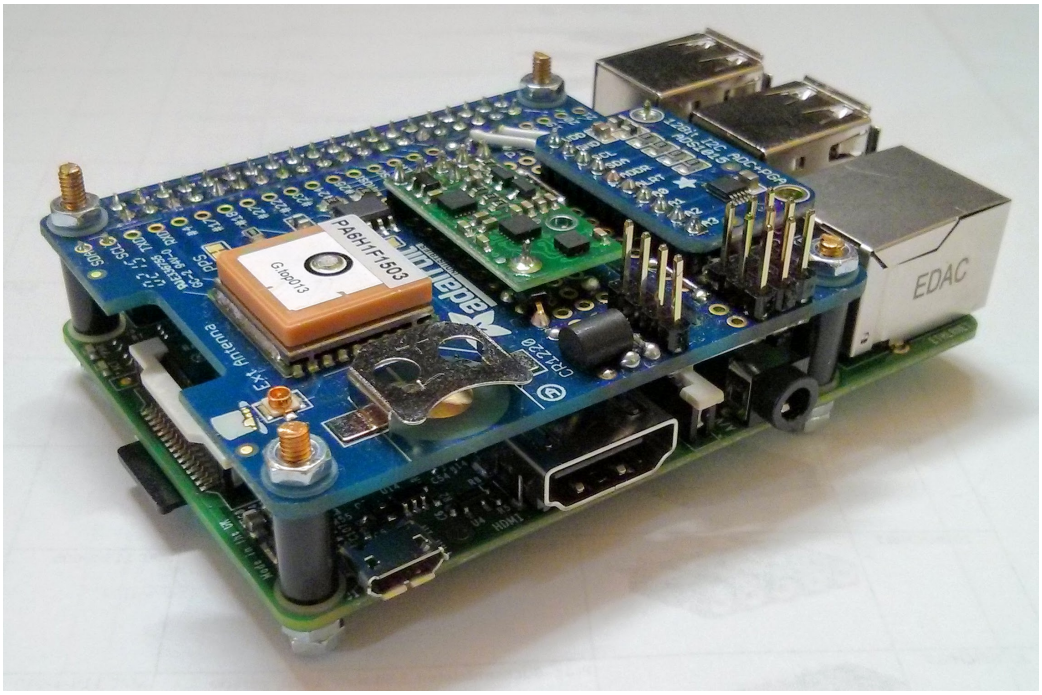


Figure 1.1: Raspberry Pi based Quadrocopter Platform (Controller Board)

1.1 Temporal project scope

The project started in **March 23, 2015** and ended in **June 26, 2015**.

The following milestones were initially set up:

Nr.	Date	Milestone's goal	Status
1	April 1, 2015	Hardware selected	**successfully reached**
2	May 11, 2015	HAL drivers finalized	**successfully reached**
3	May 25, 2015	First prototyp with flying capabilities	**partly reached**
4	June 15, 2015	First prototyp with position hold	**not reached**

Table 1.1: Initially set up milestones for the project MasterQuad 2015

1.2 Contentual project scope

The original project goals where defined as followed:

1. Select new hardware (Raspberry-based with Linux capability)
2. Setting up RTOS (Preempt RT Kernel)
3. Writing HAL drivers for new Hardware
4. Writing a sensor fusion for orientation filtering and signal enhancement
5. Integrate existing control software (if possible)
6. Setup of flight model to simulate position hold
7. Implement controllers for position hold

Not all goals could be successfully reached - items 5 to 7 could not be completely realized within the project time. Nevertheless, a stable and performant core system has been set up that can be utilized for further projects and applications.

1.3 Tasks and responsible team members

Oliver Breuning (E-Mail: olbrgs00@hs-esslingen.de)

Responsibilities:

- Virtual Machine as Development Environment (see Chapter 8)
- Low-Level-Driver for UART access
- HAL-Driver for battery check
- HAL-Driver for GPS
- HAL-Driver for Barometer
- HAL-Driver for Gyroscope
- Generic Library for fast Matrix-Operations
- Sensor-Fusion for IMU (see Chapter **??**)
- Tilt-compensation for Intertial Measurement Unit (IMU) (see Chapter **??**)
- Complementary-Filter (see Chapter **??**)
- Kalman-Filter (see Chapter **??**)

Jürgen Schmidt (E-Mail: juscgs00@hs-esslingen.de)

Responsibilities:

- Project management
- Hardware layout and mechanical packaging
- Software architecture and layout
- HAL-Driver for Accelerometer
- HAL-Driver for Magnetometer
- Live network connection for MATLAB (UDP-based)
- Linux Kernel update with PREEMPT_RT patch
- Kernel module for precise PPM measurement
- Kernel module for precise time trigger control (unstable yet!)

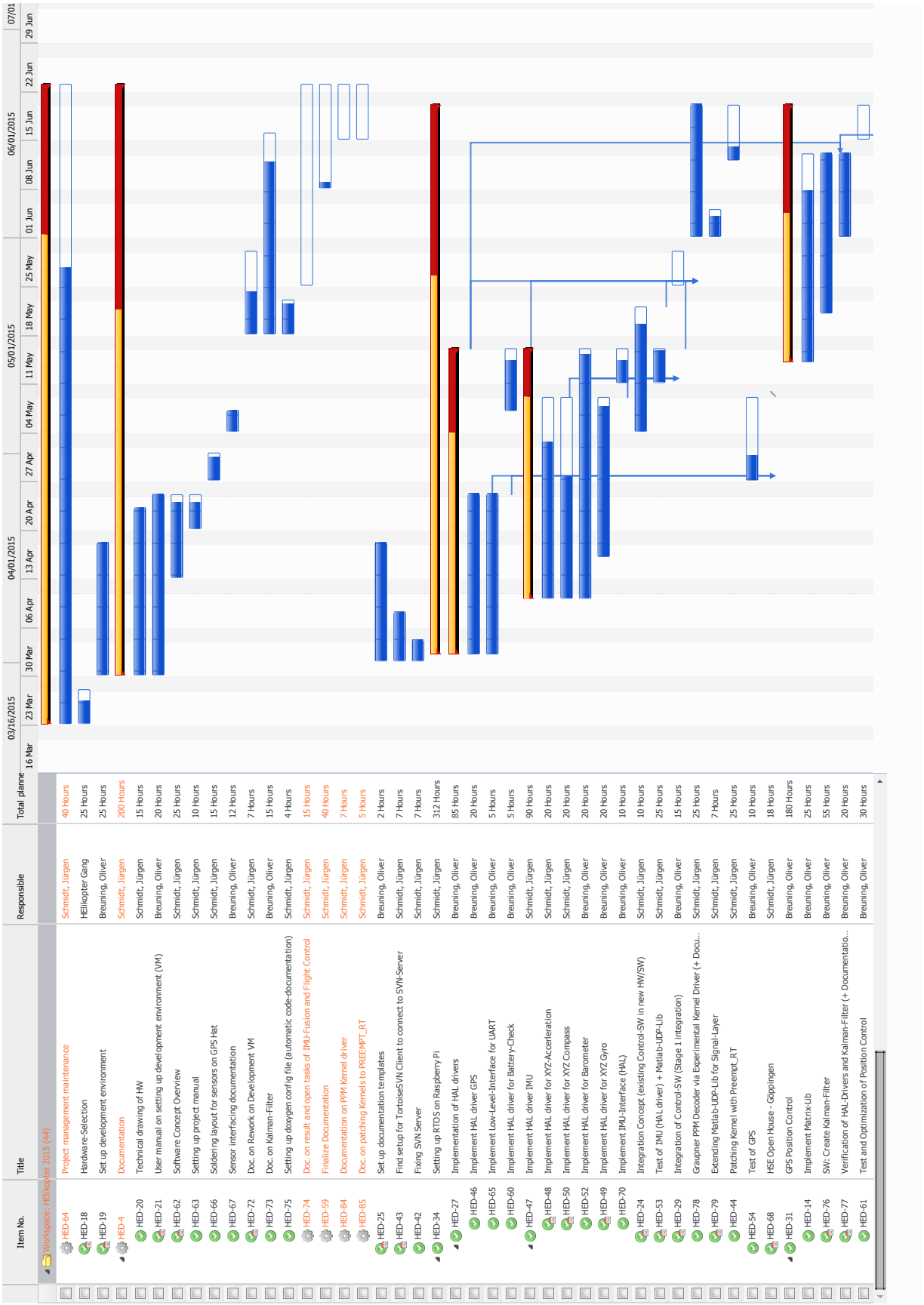


Figure 1.2: Gantt diagram of the project plan of MasterQuad 2015 (Track+ output)

1.4 Repository structure

For version controlling, the subversion system of the HElikopter Project has been used. Every team member has regarded the following given folder structure in order to keep a structured and organized working flow.

```
/
├── doc ..... Finalized documentation
│   ├── pm ..... Documentation regarding project management
│   │   ├── meetings ..... Meeting reports and meeting template
│   │   └── sources ..... LaTex sources of all documents in /doc/pm
│   └── se ..... Documentation regarding system engineering
│       └── sources ..... LaTex sources of all documents in /doc/se
├── impl ..... Source code
│   ├── branch ..... Container for parallel development lines to trunk
│   ├── tag ..... Finalized and stable software
│   └── trunk ..... Current development line
│       ├── app ..... Application Layer Software
│       ├── doc ..... Automatic code documentation via Doxygen
│       ├── hal ..... Hardware Abstraction Layer Software
│       ├── kern ..... Custom Linux Kernel modules
│       ├── matlab ..... Network connection to MATLAB
│       └── sig ..... Signal Processing Layer Software
├── scratch ..... Documents or files in progress, excluding source code
│   └── doc_template ..... Templates for documentation files (mainly LaTex)
└── sys ..... Firmwares, OS and development environment
    ├── RPi ..... Preconfigured firmware(s) for Raspberry Pi B+
    └── VM ..... Development Environment VM with all IDEs preconfigured
```

Within the folders

- `/impl/trunk/app`
- `/impl/trunk/hal`
- `/impl/trunk/sig`

a subfolder for each functional unit shall be created. The functional units shall be considered as depicted in figure 10.1 (smaller boxes inside of the colored layers). Example:

```
/
└── impl
    └── trunk
        └── hal
            ├── adc
            ├── gps
            └── gyro
```

Storing test files

For each software component, a separate test should be written to ensure the software quality. All source code files and test data files to run a test shall be saved in a separate subfolder named `tst`. Example:

```
/
└─ impl
   └─ trunk
      └─ hal
         └─ gps
            ├─ tst ..... contains data & code to test the module 'gps'
            │  ├─ test_data.dump
            │  ├─ helper_code.c
            │  ├─ helper_code.h
            │  └─ test_main.c
            ├─ gps.c ..... actual source of module 'gps'
            └─ gps.h ..... actual interfaces of module 'gps'
```

Important note:
The `tst`-folder will **not** be moved to the tags folders! All data in `/imp/tags` is considered to be well tested and stable. Therefore, the test data is not required to move.

1.5 Documentation

The project team delivered the following documents:

- Project manual (`/doc/pm`)
- Technical drawings of hardware (`/doc/se`, see also chapter 5.2)
- Bill of materials (`/doc/se`, see also chapter 5.3)
- Software structure & concept (`/doc/se`, see also chapter 10)
- User manual of project (this document)
- Doxygen-based code documentation (see appendix **??**)

The doxygen-based code documentation is delivered in the SVN repository as HTML output of the current working copy. The doxygen configuration file has been submitted in `/impl/trunk/doc/` (see also Appendix **??**). To regenerate the code documentation to the latest version, run `doxygen` in the above mentioned folder.

2 gettingStarted

This chapter will help you getting ready to work on the Quadrocopter. If you are already comfortable with the project then you might skip this chapter. This document aims at students who (like us when we started) have no clue whats going on.It will show you all the steps in order to compile the code, connect it to the Quadrocopter, and all the other stuff I dont know what we are going to to so it works. But you can find it here.

2.1 First Compiling

Lorem ipsum...

3 Landing Concept

This Chapter is about the (theoretical) ways, how to manage a autonomous landing.

In anyway we need to adjust the speed of the rotors, depending on our measurements.

3.1 Challenges

There will be several critical "challenges" during a landing process. Some of them when can estimate before and give a option, how to handle them.

3.1.1 Drift

It is very hard, to keep a Quadrocopter (in our case HElikopter) on a fixed point in a room. There is always a little bit of wind, at least from the Quadrocopter itself. Therefore you always need to correct you position, to avoid it drifting away to one side.
With our sensors implemented we can recognize the movement and take some countermeasures.

The GPS system won't work indoor, so we need to rely on the IMU (Inertial Measurement Unit). So there is no absolute measurement of the position and we need to do some calculation with the acceleration in XY-Axis, to handle the drift.

With the gyroscope we can also handle the drift about our own axis.

3.1.2 Orientation

Because of our limited sensors, and a non working GPS indoor, it will not be possible to get a absolute orientation in a room.

Outdoors we can take the GPS with the barometer, the gyroscope and the laser sensor to get a very good knowing of where we are.

3.1.3 Distance

In a steady flight situation, we measure the distance straight to the ground. But when the quadrocopter is moving, we need to calculate the distance to ground with the angle of it.

3.1.4 Landing Zone

The first step will be, to take control of the landing watching with our eyes. Just decreasing the speed of the motors (rotors) depending on the distance to ground.

When we try that the Quadrocopter takes care of a free landing zone by itself, we can think about different options:

Circling

With this method we just fly over the whole landing zone, which is a big as the quadrocopter. When we got the same distance value for the whole area, we can assume that there is no obstacle within it.

Remembering

If we take some measurements during normal flight mode, we can save that values and try to build a map of the area under us. With this method we can fly to a "free" landing zone, when the landing mode is enabled.

lateral buckling

May be the fastest and the most difficult way to land. By lateral buckling/tilting and calculation of the gathered values (depending on the angle), we can get a fast look over the area under us.

The challenge will be to stay, more or less, on the same position. Considering a certain height, where doing this action, there may be needed only small angles to tilt.

Or we could think about, mounting the laser sensor with a small angle. So to speak with a offset. We can easily consider the angle in the calculation of the height and only need to turn on the quadrocopters z-axis to check the landing zone.

3.2 Landing Mode

The Landing mode should handle the landing of the quadrocopter fully autonomous.

3.2.1 Activation

The Landing Mode will/must be activated external. With it some sensor could be activated too, or configured in a other way.

3.2.2 Abort Landing

We should be able to abort the autonomous landing, if we detect a problem.

1. approach: Just turning the Landing Mode off.
2. approach: Override the Landing Mode with the remote control.
3. approach: Abort the landing and turn the rotors off, to prevent damage to them.

4 Open Points

There are several points we wanted to do, but could not manage to do or finish them in time.

4.1 Connecting actuators

We did not connect any actuator, like an rotor, to our current software. There was no actuator-software ported to our Raspberry Pi to test with our developments.

4.2 Reacting on sensor values

There was no development to react, in any way, on our provided sensor values.
For this reason there is no controller yet, processing external signals or work between sensor and actuator.

5 Raspberry Pi based hardware platform

5.1 Hardware and device communication

5.1.1 Basic components and chosen sensors

As the main computing board the Raspberry Pi B+ has been chosen. At time of start of this project, there has been no stable Real-Time Kernel Patch (PREEMPT_RT) for the multi-core CPU Broadcom BCM2709 of Raspberry Pi 2. Fortunately, the 40-pin expansion breakout of Raspberry Pi A+/B+ and Raspberry Pi 2 are identical. In consequence, all the chosen peripheral hardware and sensors are compatible with Raspberry Pi 2. In future projects, it should be a minor task to update the Kernel of the pre-configured OS Image delivered in this project and make a Raspberry Pi 2 running with this platform. A complete list of all components with pictures can be found in chapter 5.3.

The main components chosen for this project are:

- **Main board:**
  `Raspberry Pi A+/B+` in this project. In future projects, also the `Raspberry Pi 2` is compatible with the chosen hardware as soon as a Raspberry Pi 2 compatible PREEMPT_RT Kernel Patch is available.
- **Expansion Board:**
  `Adafruit GPS Hat`, including a `GlobalTop GPS` receiver with builtin Chip-Antenna. The GPS is connected via UART to the Raspberry Pi board. In consquence, the UART may not be used for other purposes and is blocked when the GPS Hat is assembled to the Raspberry Pi!
- **Inertial Measurement Unit (IMU):**
  `Polulu AltiIMU v4 (10-DOF version)`. The Inertial Measurement Unit comprises sensors that enables to measure the pose of the Quadrocopter in space. Rotational speed, acceleration rates for each axis in space and the barometric air pressure can be sensed with the chosen IMU. The IMU is connected via I2C to the Raspberry Pi board.

  In consequence, the following sensors are included in the chosen IMU:

  - **3D Accelerometer:** `ST LSM303D` (including 3D Magnetometer/e-Compass)
  - **3D MEMS Gyroscope:** `ST L3GD20H`

  - **3D Magnetometer:** `ST LSM303D` (including 3D Accelerometer)
  - **Absolute MEMS Barometer:** `ST LPS331AP`

- **A/D Converter:**
  `Adafruit ADS1015` including ADC Chip `Texas Instruments ADS1015`. The ADC breakout board is connected via I2C to the Raspberry Pi board.
- **DC/DC Power converter:**
  `Polulu D15V70F5S3`. Since the Raspberry Pi board including all peripherals is aimed to be connected to the Li-Ion battery of the HElikopter Quadrocopters of Hochschule Esslingen, a separate DC/DC power converter has been included to the controller platform. The power converter ensures a stable power supply voltage of 5V DC at a continuous current draw up to 3.5A.

5.1.2 I2C adressing

This section shows all necessary transmissions which are needed for a successful interfacing on the I^2C bus.

- **A/D Converter**

  I^2C slave address: `0b1001001 (0x49)`

  Read

  The read command get's the data from the address, which is stored in the pointer register (blue color). See figure 5.1

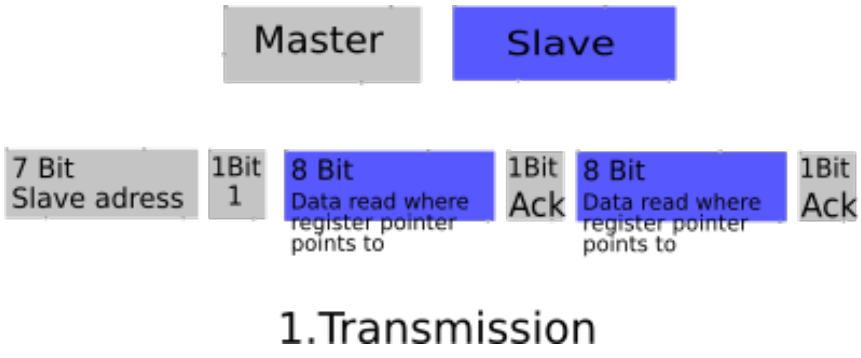


Figure 5.1: Scheme to read data from the ADC

Write

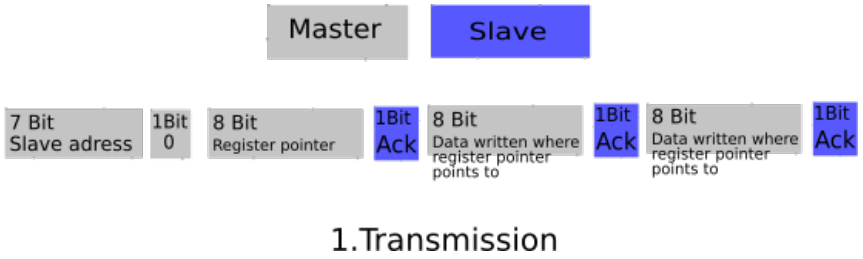


Figure 5.2: Scheme to write data to the ADC

Read conversion register

To enable a read from a conversion register, several packages need to be sent. They can be seen in figure 5.3. All slave and master acknowledges are not shown because they are handled direct by the interface and so not important for the application.

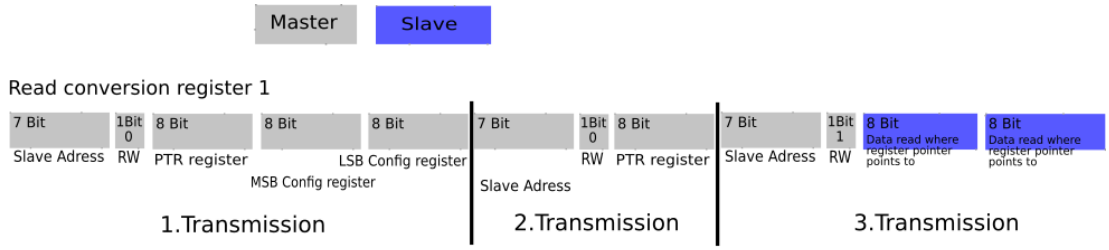


Figure 5.3: Transmission scheme to read the ADC's conversion registers

- **Inertial Measurement Unit (IMU)**

  The Inertial measurement unit (IMU) has three different chips mounted. Each chip solves one of the measurements of this unit. Each chip has a different I^2C address. All slave and master acknowledges are not shown because they are handled direct by the interface and are not important to the application level.

  - **Acceleration and Magnet Sensor**

    I^2C slave address: `0b0011110 (0x1E)`

    There are several registers which have to be configured before reading and also several register where the acceleration, magnetic strength and if needed temperature can be read. To reduce the amount of pages of this document, they will be not listed here. All the registers can be found in the datasheet `IMU_LSM303D.pdf`, stored in the SVN directory `/doc/se/Datasheets/IMU`.

Read

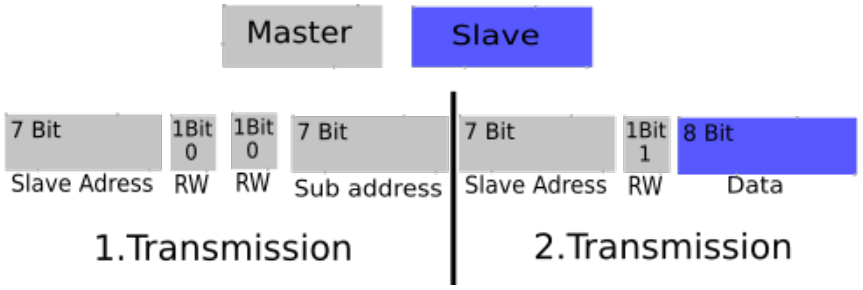


Figure 5.4: Transmission scheme for a single byte read of the ACC-Sensor

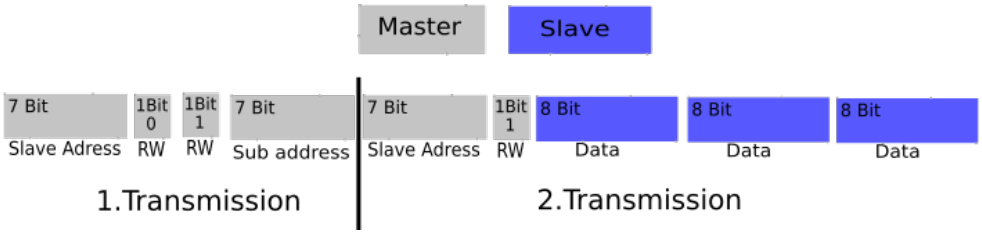


Figure 5.5: Transmission scheme for multiple data read of the ACC-Sensor(burst read)

1. **Transmission**: Slave address including RW bit ('0'): `0x3C`
2. **Transmission**: Slave address including RW bit ('1'): `0x3D`

Write

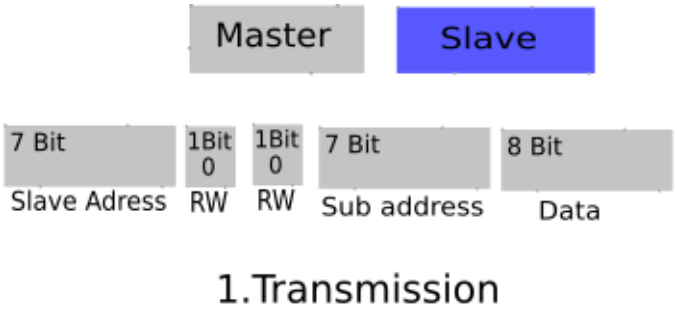


Figure 5.6: Transmission scheme for a single byte write of the ACC-Sensor

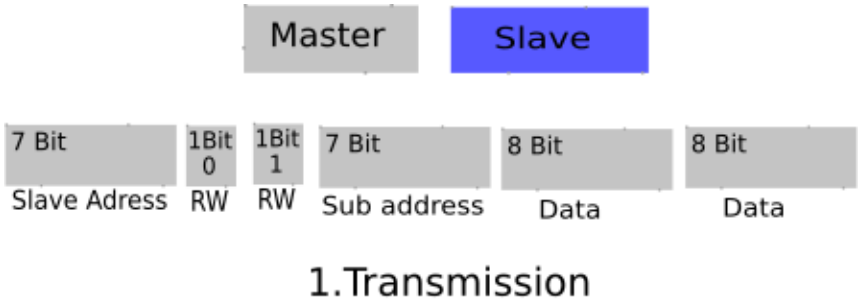


Figure 5.7: Transmission scheme for multiple data write of the ACC-Sensor(burst write)

1.Transmission: Slave address including RW bit ('0'): `0x3C`

– **Gyroscope Sensor**

I²C slave address: `0b1101010 (0x6A)`

There are several registers which have to be configured before reading and also several register where the rotational speed and if needed the temperature can be read. To reduce the amount of pages of this document, they will be not listed here. All the registers can be found in the datasheet `IMU_L3GD20H.pdf`, which is stored in the SVN directory `\doc\se\Datasheets\IMU`.

Read

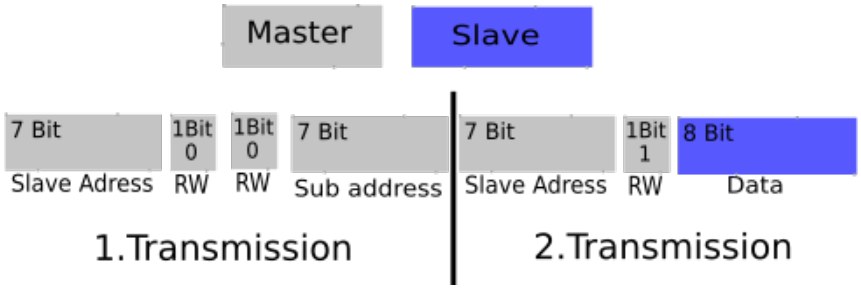


Figure 5.8: Transmission scheme for a single byte read of the Gyro-Sensor

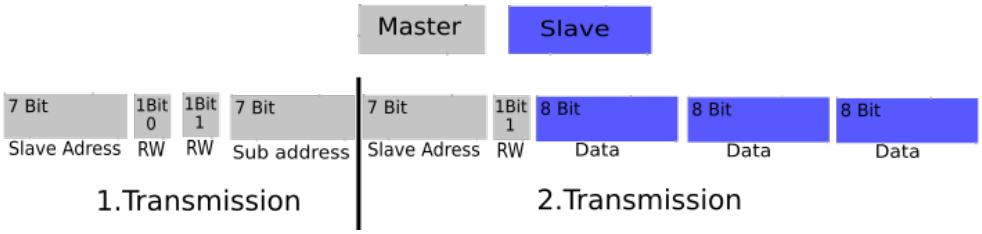


Figure 5.9: Transmission scheme for multiple data read of the Gyro-Sensor (burst read)

1.Transmission: Slave address including RW bit ('0'): `0xD4`
2.Transmission: Slave address including RW bit ('1'): `0xD5`

Write

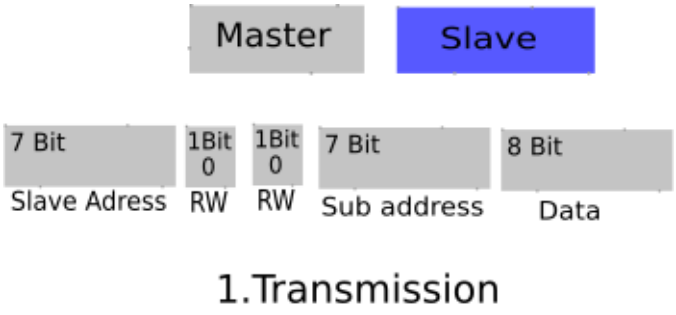


Figure 5.10: Transmission scheme for a single byte write of the Gyro-Sensor

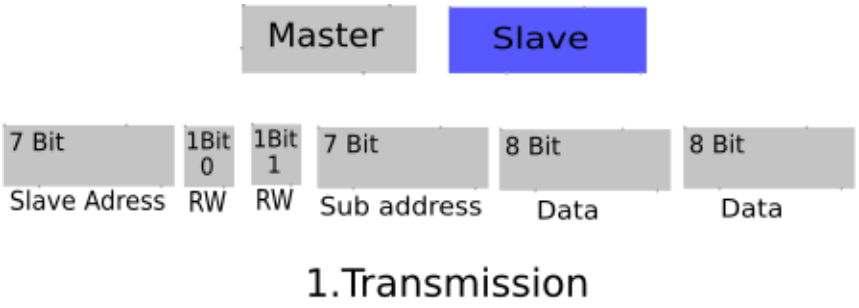


Figure 5.11: Transmission scheme for multiple data write of the Gyro-Sensor (burst write)

1.Transmission: Slave address including RW bit ('0'): `0xD4`

– **Pressure Sensor**

I²C slave address: `0b1011100 (0x5C)`

There are several registers which have to be configured before reading and also several register where the pressure and if needed the temperature can be read. To reduce the amount of pages of this document, they will be not listed here. All the registers can be found in the datasheet `IMU_LPS331AP.pdf`, which is stored in the SVN directory `\doc\se\Datasheets\IMU`.

Read

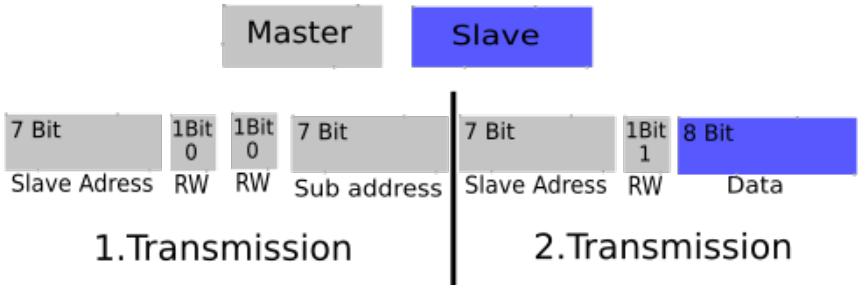


Figure 5.12: Transmission scheme for a single byte read of the Pressure-Sensor

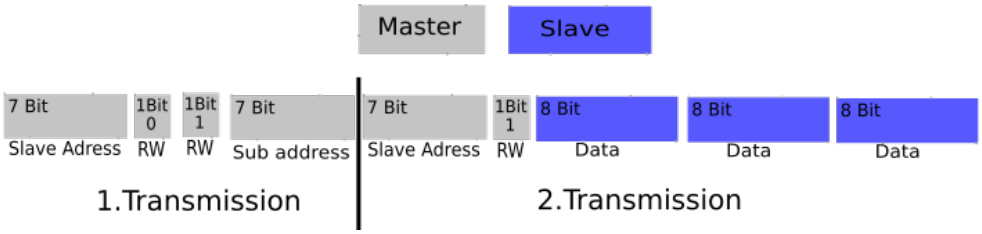


Figure 5.13: Transmission scheme for multiple data read of the Pressure-Sensor(burst read)

1.Transmission: Slave address including RW bit ('0'): `0xB8`
2.Transmission: Slave address including RW bit ('1'): `0xB9`

Write

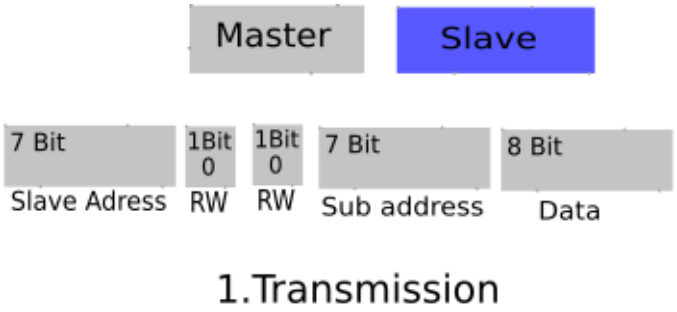


Figure 5.14: Transmission scheme for a single byte write of the Pressure-Sensor

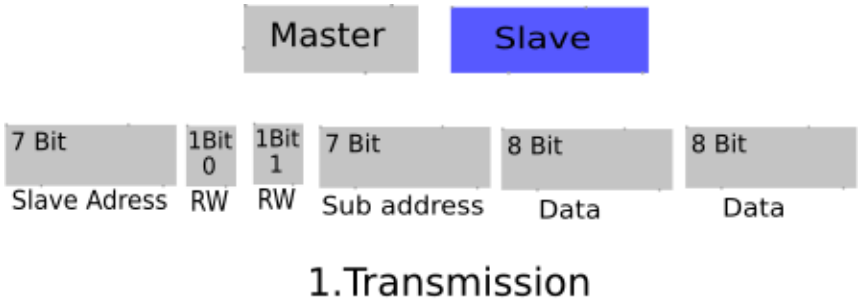


Figure 5.15: Transmission scheme for multiple data write of the Pressure-Sensor(burst write)

1.Transmission: Slave address including RW bit ('0'): 0xB8

- **Motor Driver**

  All slave and master acknowledges are not shown because they are handled direct by the interface and so not important here. To enable flying with a Quadrocopter there are four motors and so four brushless drivers needed. Each of them has an individual address.

  I²C slave addresses:
  Motor 1 –> `0b0101001 (0x29)`
  Motor 2 –> `0b0101010 (0x2A)`
  Motor 3 –> `0b0101011 (0x2B)`
  Motor 4 –> `0b0101100 (0x2C)`

5.1.3 Read

Read operations: NOT DEFINED

5.1.4 Write

Write operations:

Possible data values are in the range of `10` (Decimal) up to `255` (Decimal) which refers to a hexadecimal range of `0x0A` to `0xFF`.

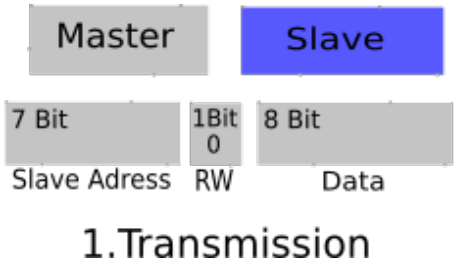


Figure 5.16: Transmission scheme for a single byte write to a motor driver board

1.Transmission: Slave address including RW bit ('0'):
Motor 1 –> `0x52`
Motor 2 –> `0x54`
Motor 3 –> `0x56`
Motor 4 –> `0x58`

Possible Data values are in the range of 10 (Decimal) up to 255 (Decimal). So in the range from 0x0A to 0xFF.

5.1.5 UART communication

The GPS module uses the UART interface of the Raspberry Pi. The used GPS module has a fixed communication set up.
To ensure working the following values need to be used:

- Baudrate: 9600 baud
- Databits: 8 Bits
- Stopbits: 1 Bit
- Modem control: No

5.2 Technical drawings and packaging

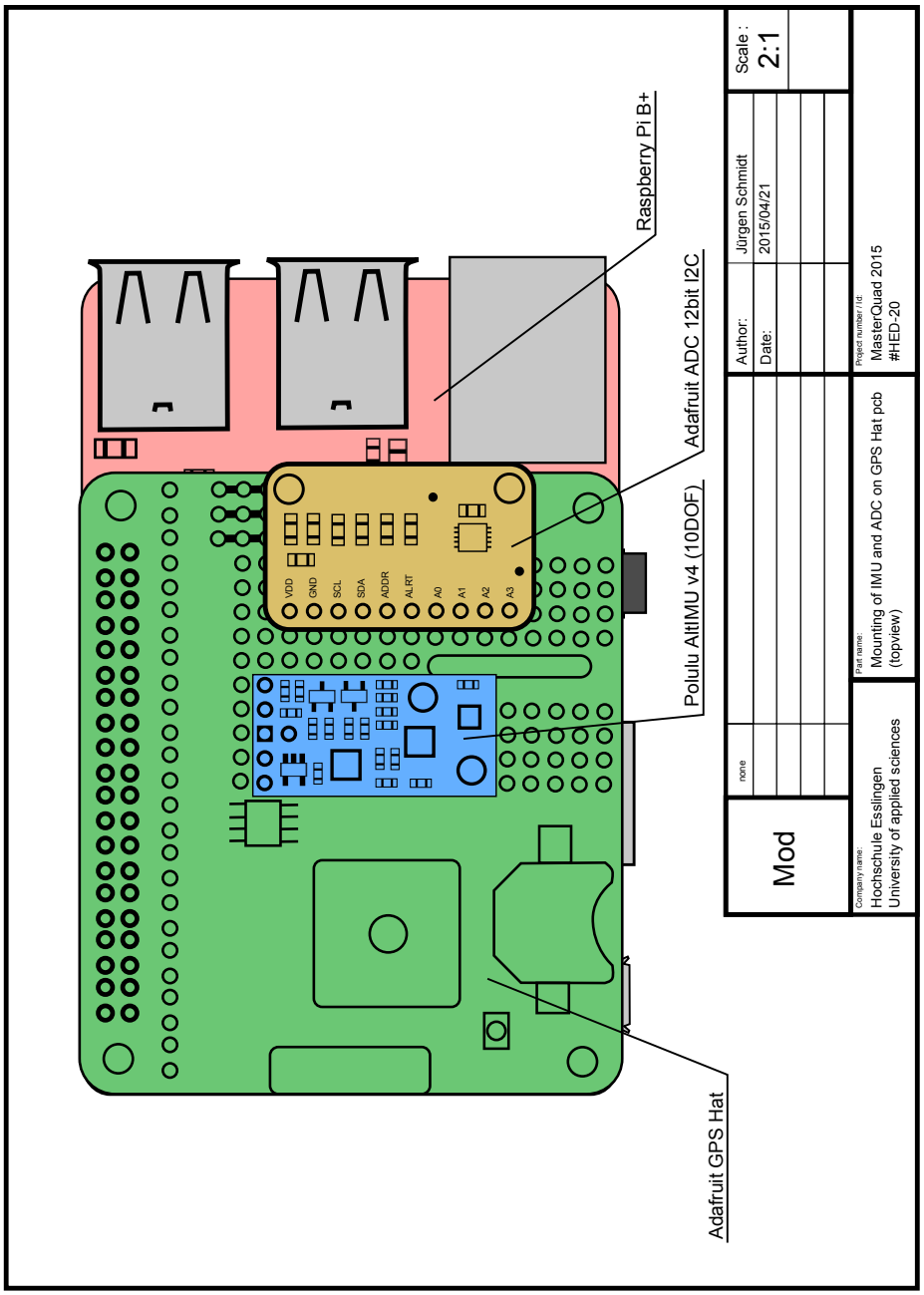


Figure 5.17: Placing ADC and IMU on GPS Hat. The drawing shows the exact position of the sensor boards, aligned with matching soldering pads of the GPS Hat and the Sensor baords.

To mount the assembled GPS Hat on top of the Raspberry Pi (as depicted in fig. 5.18) the following parts are required:

- 4 pcs. polyamid standoffs, 10mm height (Bürklin, 18H5045)
- 4 pcs. threaded rod, M2.5 in 40mm pieces (Bürklin, 16H322)
- 16 pcs. polyamid shims, M2.5 (Bürklin, 16H942)
- 12 pcs. hexagonal nuts, M2.5 (Bürklin, 16H722)
- 1 pc. GPS Hat with assembled sensors/peripherals as shown in fig. 5.17 and soldered according to 5.19 and fig. 5.20
- 1 pc. Raspberry Pi A+/B+/2

All referenced parts are also mentioned in the Bill of Materials in Chapter 5.3.

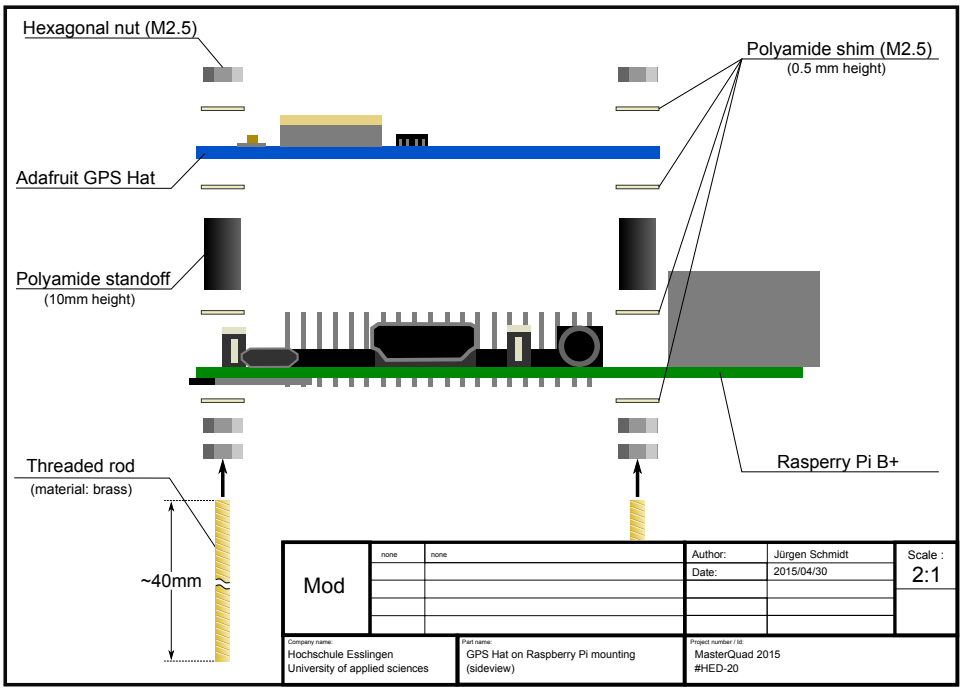


Figure 5.18: Stacked mounting of Hat-Board on Raspberry Pi

To ease the soldering of the IMU and ADC on top of the GPS Hat board, a soldering layout has been created as shown in fig. 5.19 and fig. 5.20. The layout shows in a one-by-one view the soldering pads of the GPS Hat (one view onto the top, one view onto the bottom). According to the experience of the authors of this document, the soldering layout figures eases the soldering of the sensors to the GPS Hat for students, that are not used to solder a densely packed PCB.

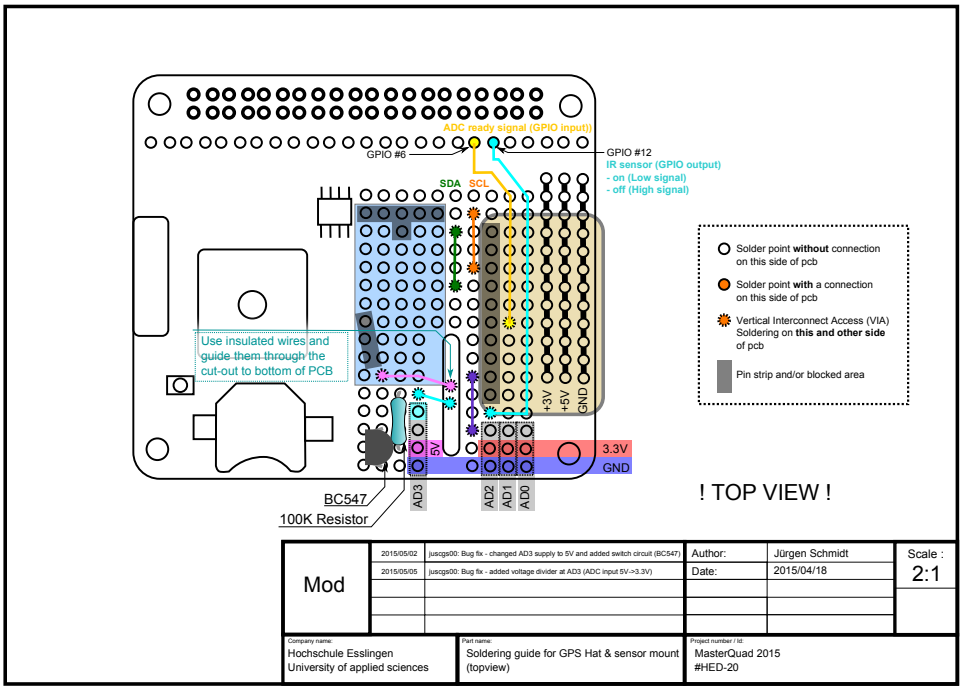


Figure 5.19: Soldering plan for Raspberry Pi Hat (top view)

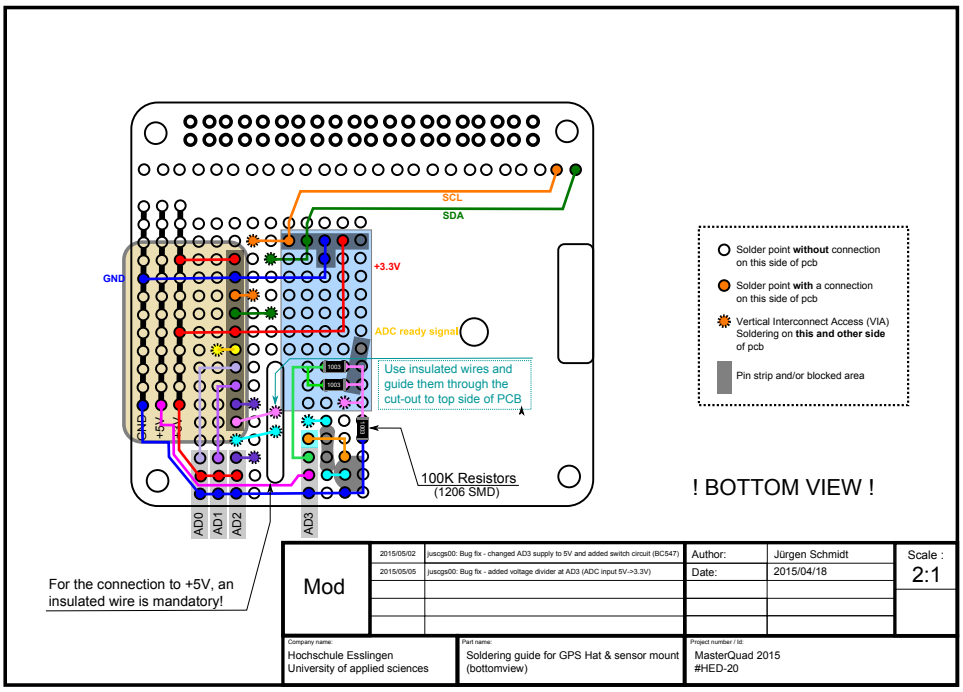


Figure 5.20: Soldering plan for Raspberry Pi Hat (bottom view)

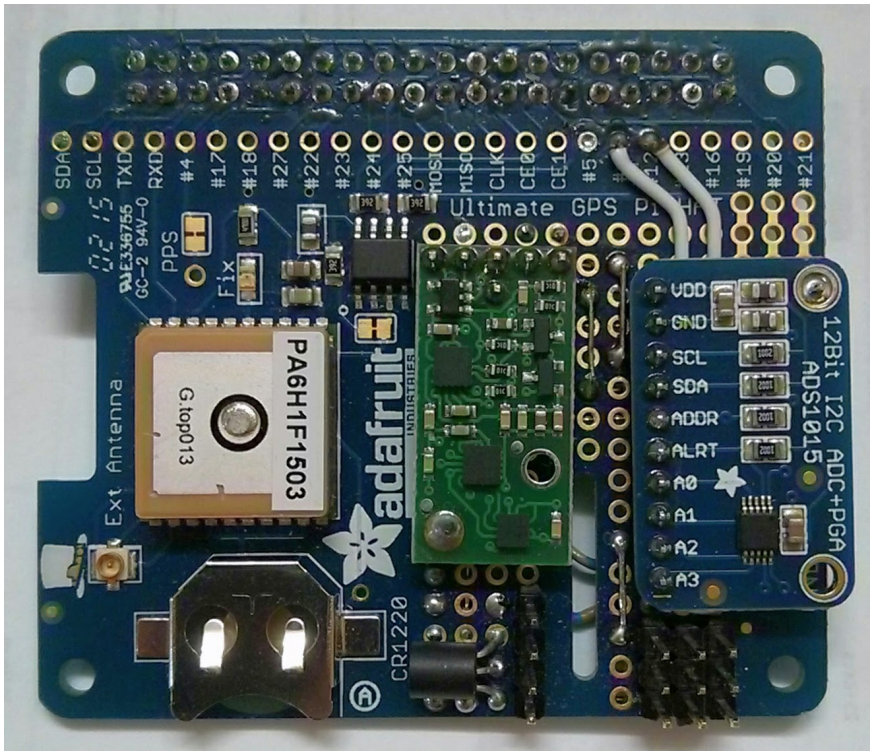


Figure 5.21: Soldered Raspberry Pi Hat (top view)

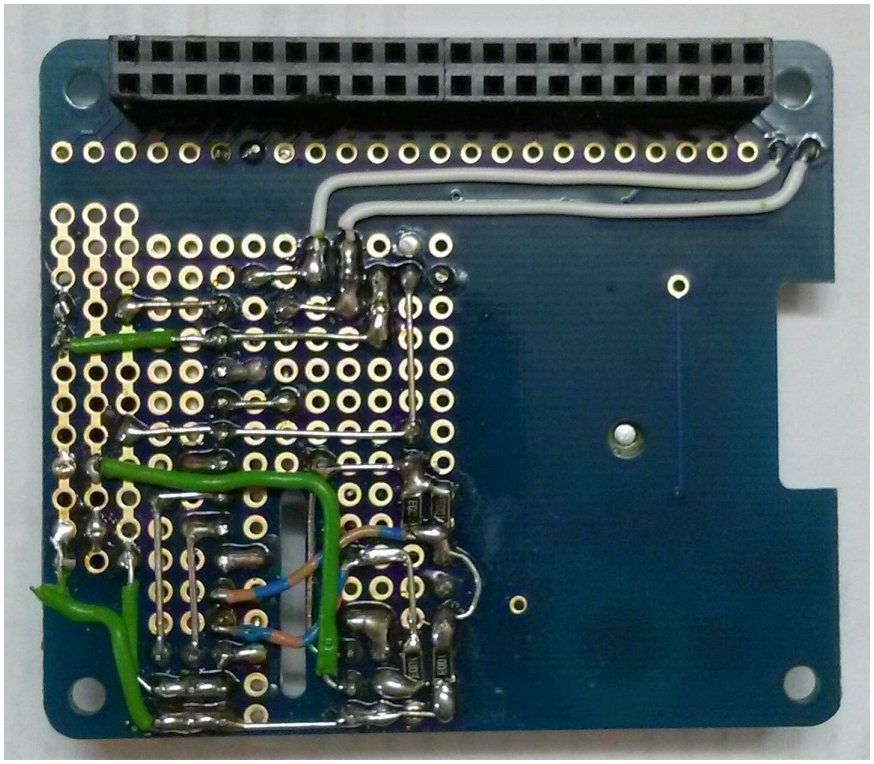


Figure 5.22: Soldered Raspberry Pi Hat (bottom view)

5.3 Bill of materials

For this project, the parts nedded for the Raspberry Pi Platform has been ordered 5 times. The following Bill of Materials lists all parts including the distributor and parts numbers.

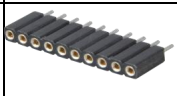
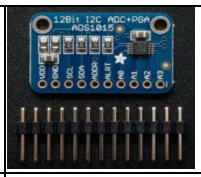
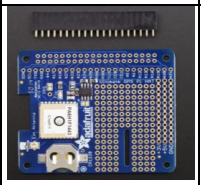
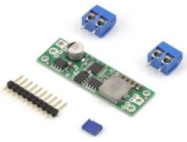
#	Shop	Picture	Description	Art.-Nr.	Price per Unit	Total price	Link
5	REICHELT		Raspberry Pi B+ (All-In Package) including: > Power supply (Micro-USB Stecker, 5V/1.2A) > MicroSD-Card 8GB > Housing for Rasp Pi B+ > WiFi USB-Dongle (EDIMAX 150MBit/s)	RASP B+ ALL IN	59,95 €	299,75 €	http://www.reichelt.de/Einplatinen-Computer/RASP-B-ALL-IN/3/index.html?&ACTION=3&LA=3&ARTICLE=148139&GROUPID=6666
5	REICHELT		HDMI-auf-DVI Cable	AK HDMI-DVI 2,0	3,80 €	19,00 €	http://www.reichelt.de/AK-HDMI-DVI-2-0/3/index.html?&ACTION=3&LA=446&ARTICLE=69657&artnr=AK+HDMI-DVI+2%2C0&SEARCH=hdmi+auf+dvi
5	REICHELT		Cable USB 2.0A to Micro-USB B (1,8m)	AK 676-AB2	1,95 €	9,75 €	http://www.reichelt.de/USB-Kabel/AK-676-AB2/3/index.html?&ACTION=3&LA=3&ARTICLE=133000&GROUPID=6099
5	REICHELT		Cross-Over-Kabel, doppelt geschirmt, 2 Meter	PATCHKABEL-X 2	1,20 €	6,00 €	http://www.reichelt.de/Patchkabel-Netzwerkkabel-cross-over/PATCHKABEL-X-2/3/index.html?&ACTION=3&LA=2&ARTICLE=25802&GROUPID=5848&artnr=PATCHKABEL-X+2
10	REICHELT		Stiftleisten 2,54 mm, 1X04, gerade	MPE 087-1-004	0,10 €	1,00 €	http://www.reichelt.de/Stiftleisten/MPE-087-1-004/3/index.html?&ACTION=3&LA=2&ARTICLE=119881&GROUPID=3220&artnr=MPE+087-1-004
20	REICHELT		Stiftleisten 2,54 mm, 1X05, gerade	MPE 087-1-005	0,12 €	2,40 €	http://www.reichelt.de/Stiftleisten/MPE-087-1-005/3/index.html?&ACTION=3&LA=2&ARTICLE=119882&GROUPID=3220&artnr=MPE+087-1-005
10	REICHELT		Stiftleisten 2,54 mm, 1X10, gerade	MPE 087-1-010	0,25 €	2,50 €	http://www.reichelt.de/Stiftleisten/MPE-087-1-010/3/index.html?&ACTION=3&LA=2&ARTICLE=119885&GROUPID=3220&artnr=MPE+087-1-010
10	REICHELT		Stiftleisten 2,54 mm, 1X20, gerade	MPE 087-1-020	0,33 €	3,30 €	http://www.reichelt.de/Stiftleisten/MPE-087-1-020/3/index.html?&ACTION=3&LA=2&ARTICLE=119888&GROUPID=3220&artnr=MPE+087-1-020
10	REICHELT		Buchsenleisten 2,54 mm, 1X04, gerade	MPE 094-1-004	0,19 €	1,90 €	http://www.reichelt.de/Buchsenleisten/MPE-094-1-004/3/index.html?&ACTION=3&LA=2&ARTICLE=119913&GROUPID=3221&artnr=MPE+094-1-004
10	REICHELT		Präz.-Buchsenleisten 2,54 mm, 1X05, gerade	MPE 115-1-005	0,35 €	3,50 €	http://www.reichelt.de/Buchsenleisten/MPE-115-1-005/3/index.html?&ACTION=3&LA=2&ARTICLE=119953&GROUPID=3221&artnr=MPE+115-1-005
10	REICHELT		Präz.-Buchsenleisten 2,54 mm, 1X10, gerade	MPE 115-1-010	0,71 €	7,10 €	http://www.reichelt.de/Buchsenleisten/MPE-115-1-010/3/index.html?&ACTION=3&LA=2&ARTICLE=119955&GROUPID=3221&artnr=MPE+115-1-010
5	REICHELT		USB2.0 Card Reader All-in-1	DELOCK 91471	6,15 €	30,75 €	http://www.reichelt.de/DELOCK-91471/3/index.html?&ACTION=3&LA=446&ARTICLE=106309&artnr=DELOCK+91471&SEARCH=sd+card+reader

Figure 5.23: Bill of materials, part 1

5	EXP-Tech		4channel 12bit (3.3kSPS) ADC over I2C	EXP-R15-038	9,25 €	46,25 €	http://www.exp-tech.de/adafruit-ads1015-12-bit-adc-4-channel-with-progrmmable-gain-amplifier
5	EXP-Tech		Pololu AltIMU-10 v4	EXP-R25-371	26,61 €	133,05 €	http://www.exp-tech.de/pololu-altimu-10-v4-gyro-accelerometer-compass-and-altimeter-l3gd20h-lsm303d-and-lps25h-carrier
5	EXP-Tech		Adafruit Ultimate GPS HAT	EXP-R15-665	53,50 €	267,50 €	http://www.exp-tech.de/adafruit-ultimate-gps-hat-for-raspberry-pi-a-or-b-mini-kit
10	EXP-Tech		GPIO Header for Raspberry Pi B+ (stackable)	EXP-R15-621	2,50 €	25,00 €	http://www.exp-tech.de/gpio-header-for-raspberry-pi-b-extra-long-2x20-female-header
5	EXP-Tech		USB to TTL Serial Cable (GPS-Prototyping)	EXP-R15-015	9,47 €	47,35 €	http://www.exp-tech.de/usb-to-ttl-serial-cable-debug-console-cable-for-raspberry-pi?___SID=U
5	EXP-Tech		PULSEDLight LIDAR Lite (Distance-Measurement Unit)	EXP-R05-729	94,90 €	474,50 €	http://www.exp-tech.de/lidar-lite
5	EXP-Tech		Pololu Step-Down Spannungsregler D15V70F5S3	EXP-R25-037	22,80 €	114,00 €	http://www.exp-tech.de/pololu-step-down-spannungsregler-d15v70f5s3

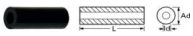
2	Bürklin		Gewindestangen DIN 975 Messing Typ BS. M2.5 (1000mm)	16 H 322	5,21 €	10,42 €	https://www.buerklin.com/default.asp?event=ShowArtikel(16H322)&context=subset:0;sel:SE;search:gewindestange;se_name:vt;patchid:;tags:;pagecount:100&l=d&jump=ArtNr_16H322&ch=25059
1	Bürklin		Schrauben-Sicherungsmittel Typ Ergo 4003/4052/4101, mittelfest (10g)	12 L 585	6,60 €	6,60 €	https://www.buerklin.com/default.asp?event=ShowArtikel(12L585)&context=subset:0;sel:SE;search:Typ%C2%A0Ergo%C2%A04003/4052/4101;se_name:vt;patchid:;tags:;pagecount:100&l=d&jump=ArtNr_12L585&ch=69315
1	Bürklin		Sechskantmuttern Mat. 4.8-Zn DIN 934/ISO 4032 Typ BS 3200, Gew. M 2,5 (100 Stk.)	16 H 722	1,67 €	1,67 €	https://www.buerklin.com/default.asp?event=ShowArtikel(16H722)&context=subset:0;sel:P;search:Sechskantmuttern;patchid:;tags:Sechskantmuttern;pagecount:100&l=d&jump=ArtNr_16H722&ch=61185
50	Bürklin		Distanzrollen aus Polyamid Typ Fastpoint 10 (10090BB0113.5, M 2,5, L 13,5, Ad 5,0, Id 2,7 mm)	18 H 5045	0,08 €	4,00 €	https://www.buerklin.com/default.asp?event=ShowArtikel(18H5045)&context=subset:0;sel:P;search:Distanzh%C3%BClsen;patchid:;tags:Distanzh%C3%BClsen;pagecount:100&l=d&jump=ArtNr_18H5045

Figure 5.24: Bill of materials, part 2

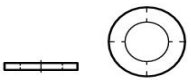
50	Bürklin		Distanzrollen aus Polyamid Typ Fastpoint 10 (10090BB0110.0, M 2,5, L 10, Ad 5,0, Id 2,7 mm)	18 H 5044	0,07 €	3,50 €	https://www.buerklin.com/default.asp?event=ShowArtikel(18H5044)&context=subset:0;sel:P;search:Distanzh%C3%BClsen;patchid:;tags:Distanzh%C3%BClsen;pagecount:100&l=d&jump=ArtNr_18H5044&ch=74532
1	Bürklin		Unterlegscheiben aus PA 6.6 DIN 125/ISO 7089 Typ 05., M2.5 (100 Stk.)	16 H 942	2,83 €	2,83 €	https://www.buerklin.com/default.asp?event=ShowArtikel(16H942)&context=subset:0;sel:SE;search:unterlegschreiben;se_name:vt;patchid:;tags:;pagecount:100&l=d&jump=ArtNr_16H942&ch=10217
				Summe		**1.523,62 €**	
				Price per Kit		304,72 €	

Figure 5.25: Bill of materials, part 3

6 Individual parts

6.1 Raspberry Pi B+

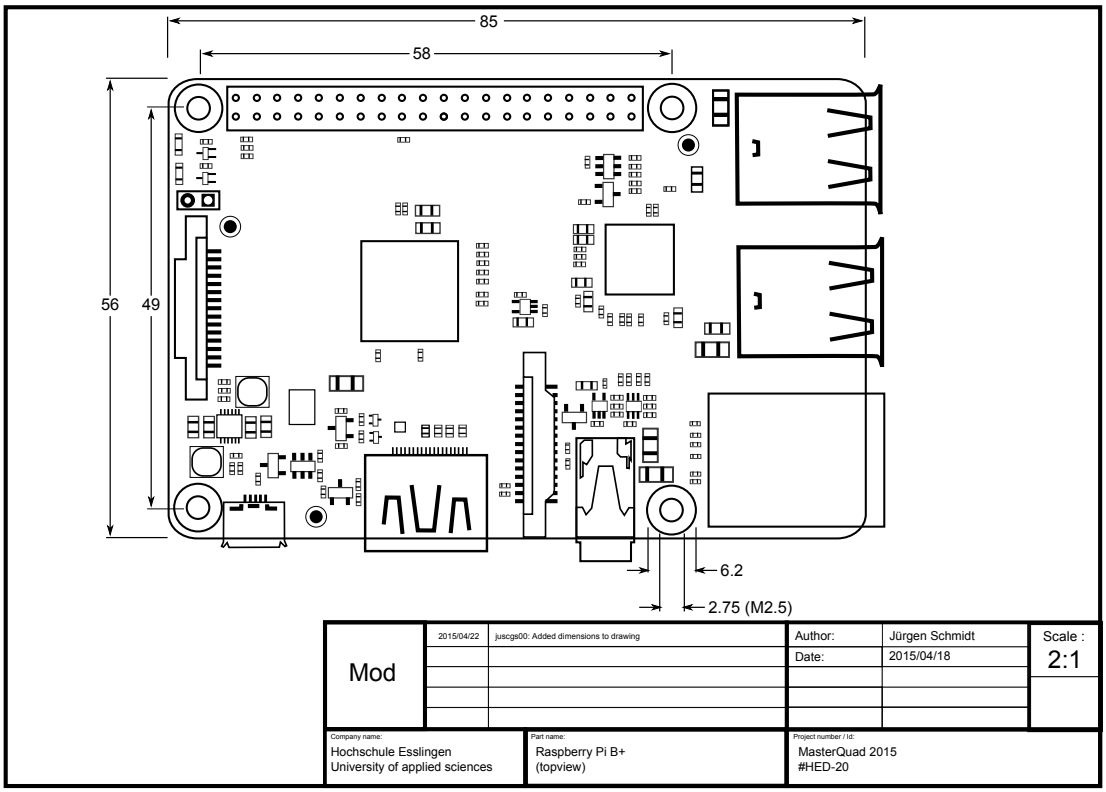


Figure 6.1: Raspberry Pi B+ (topview)

6.2 Adafruit GPS Hat

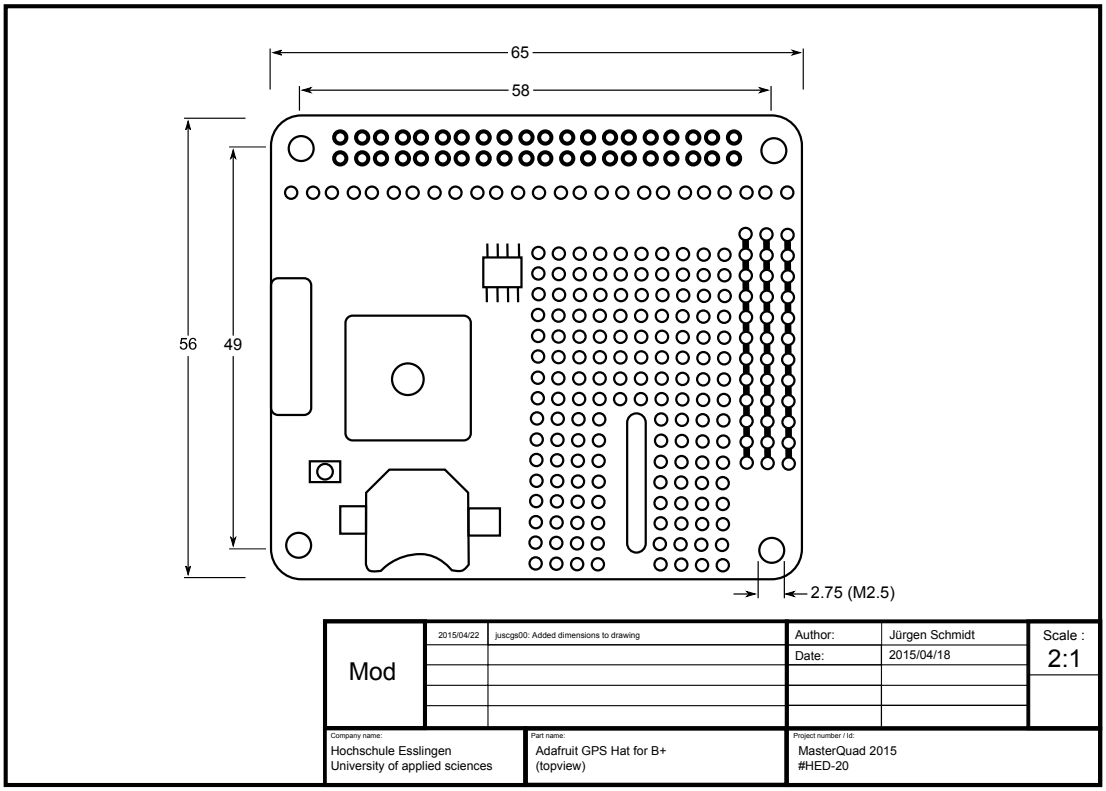


Figure 6.2: Adafruit GPS Hat

6.3 Polulu AltIMU v4

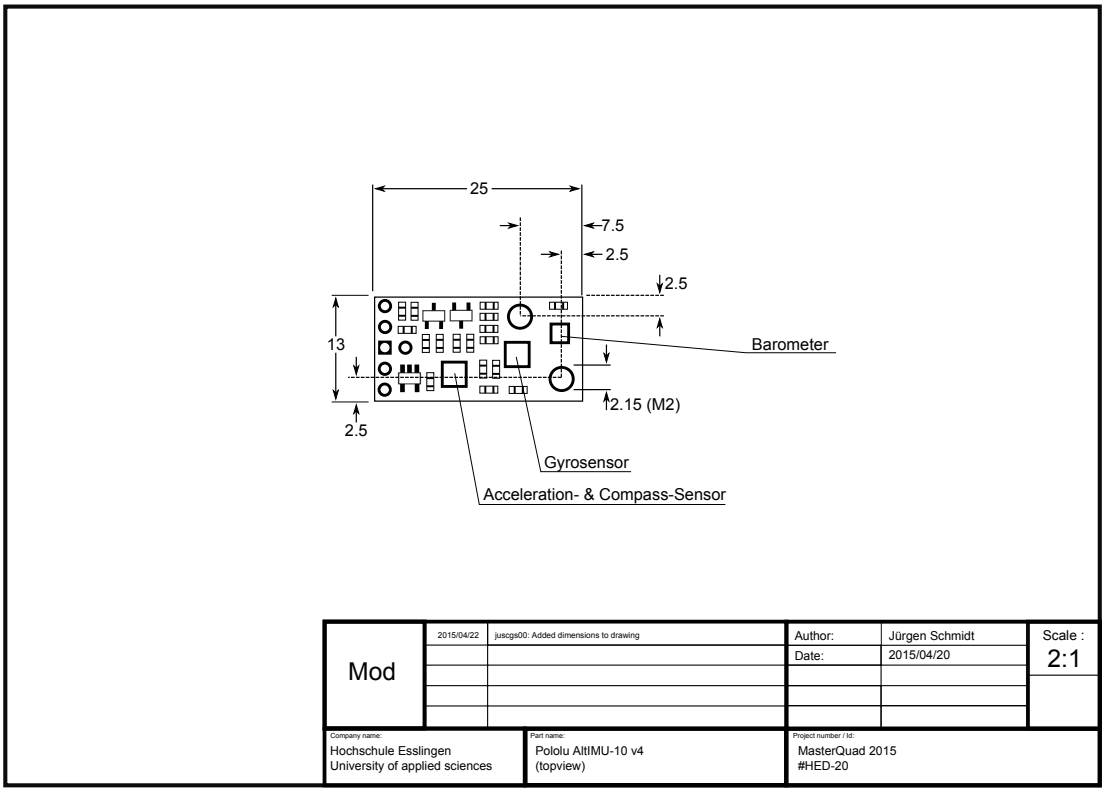


Figure 6.3: Polulu AltIMU v4

6.4 Adafruit 12bit ADC over I2C

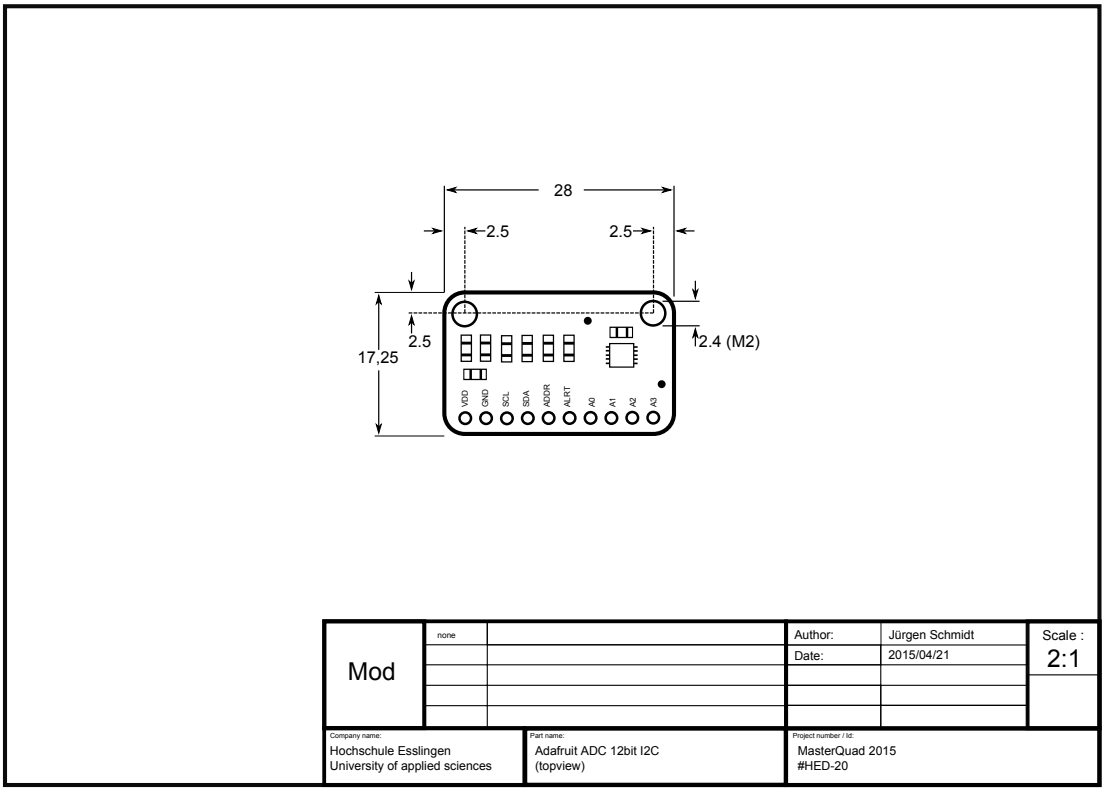


Figure 6.4: Adafruit 12bit ADC over I2C

7 Assembly

7.1 Placing ADC and IMU on GPS Hat

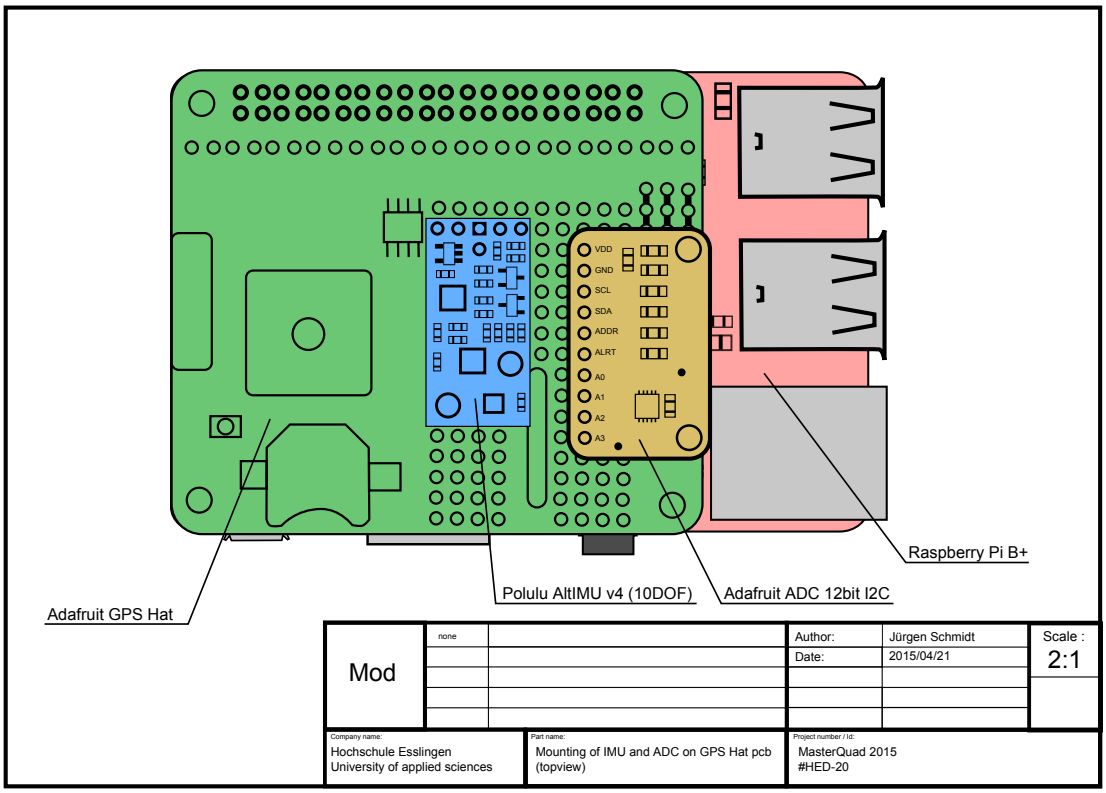


Figure 7.1: Placing ADC and IMU on GPS Hat

7.2 Wedding of GPS Hat and Raspberry Pi B+

8 Development Environment (Virtual Machine Image and Native Machine)

This project uses the VMWare player of the company VMWare [1]. This gives the ability that no dependency to a special operating system exists. It can run on any system like mac, windows or linux. A pre defined Image for the VMWare player can be found in the SVN repository folder `/sys/VM/`. The native machine is based on 14.04 AMD64 Ubuntu. If you have native machine and running the program there, then please directly skip to the section **Toolchain**

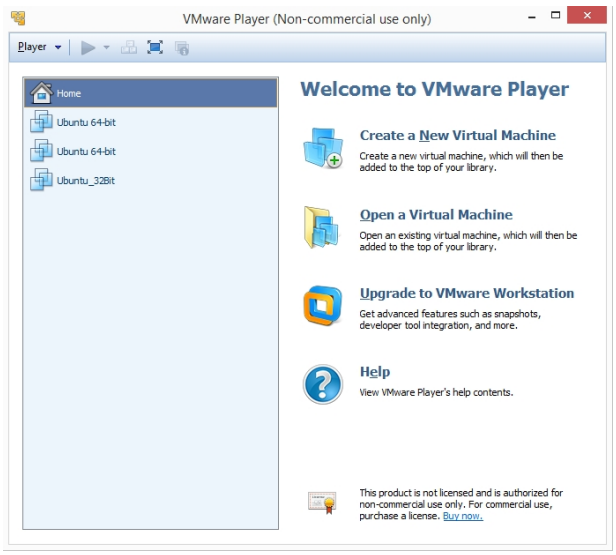


Figure 8.1: VMWare Player

8.1 Ubuntu

Here the current Ubuntu 14.10 which is installed with the help of the VMWare player is used. For programming the software eclipse is taken. First the current image of Ubuntu needs to be downloaded:

1 https://my.vmware.com/de/web/vmware/free#desktop_end_user_computing/vmware_player/7_0

```
1 http://www.ubuntu.com/download
```

Figure 8.2 shows the the window of the VMWare player directly after starting. To add a new virtual machine a click on 'Create a New Virtual Machine' has to be made.

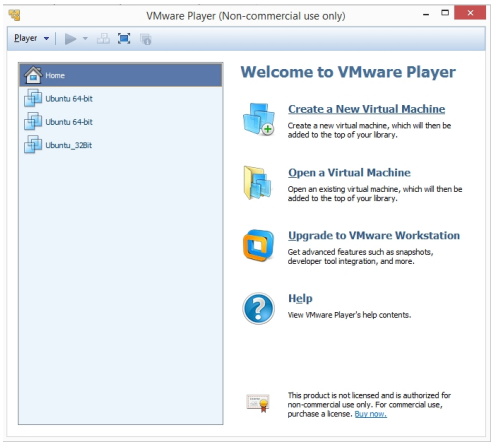


Figure 8.2: Installing Ubuntu - Step 1: Create new virtual machine

Then under the 'Installer disc image file (iso)', the path to the before downloaded ISO-file needs to be set like in the figure 8.3.

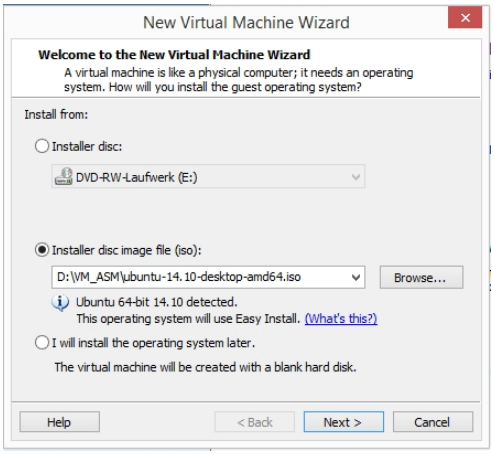


Figure 8.3: Installing Ubuntu - Step 2: Installer ISO-file

Figure 8.4 shows the next step. A full name, a user and a password is set. In this project 'user' with password 'user' is used.

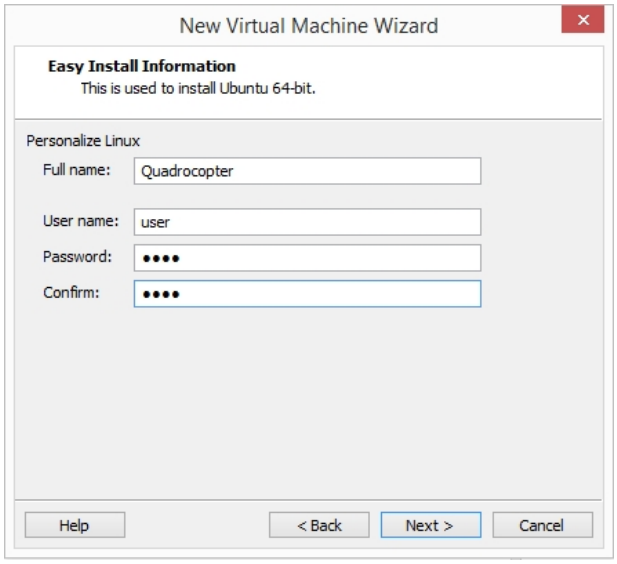


Figure 8.4: Installing Ubuntu - Step 3: Setting up a generic user

Then a name for the virtual machine and the location, where the virtual machine is stored is chosen. See figure 8.5.

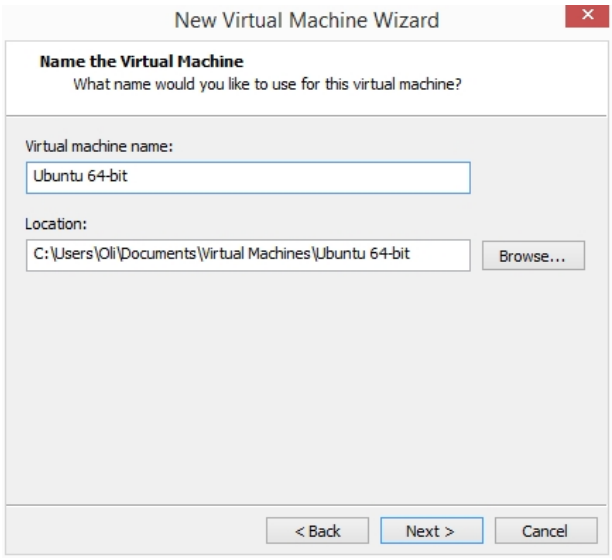


Figure 8.5: Installing Ubuntu - Step 4: Name of virtual machine

The following figures 8.6 and 8.7 show the option for the size and the splitting and finally a summary.

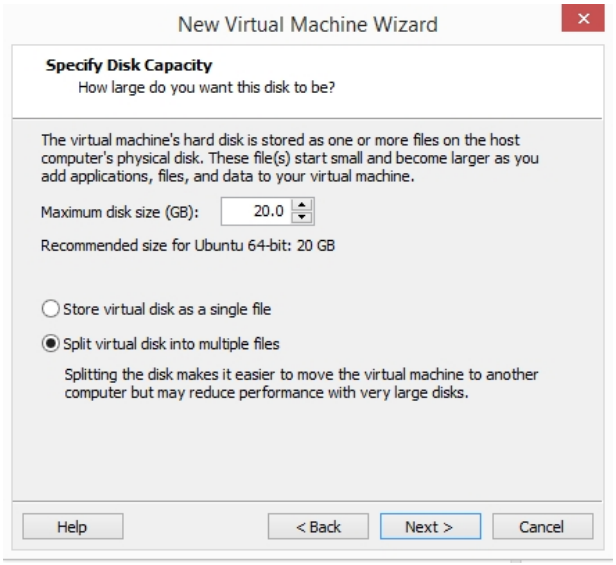


Figure 8.6: Installing Ubuntu - Step 5: Choosing virtual disk size

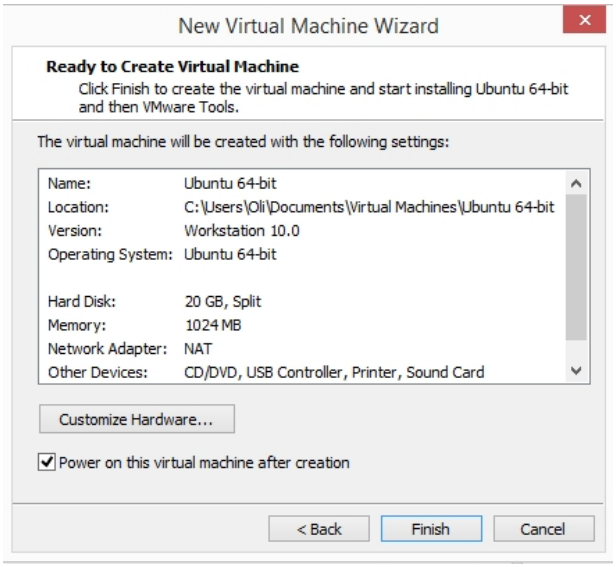


Figure 8.7: Installing Ubuntu - Step 6: Finalize virtual machine

Then VMWare tools wants to be installed. This tool is needed for example to enable dynamic rescaling of the Desktop of Ubuntu within the VMWare player, so it should be installed. See figure 8.8.

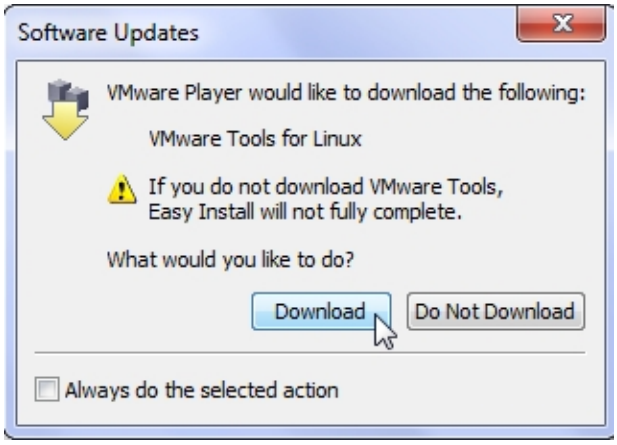


Figure 8.8: Installing Ubuntu - Step 7: Installation of VMWare Tools

Finally the login window of Ubuntu (figure 8.9).

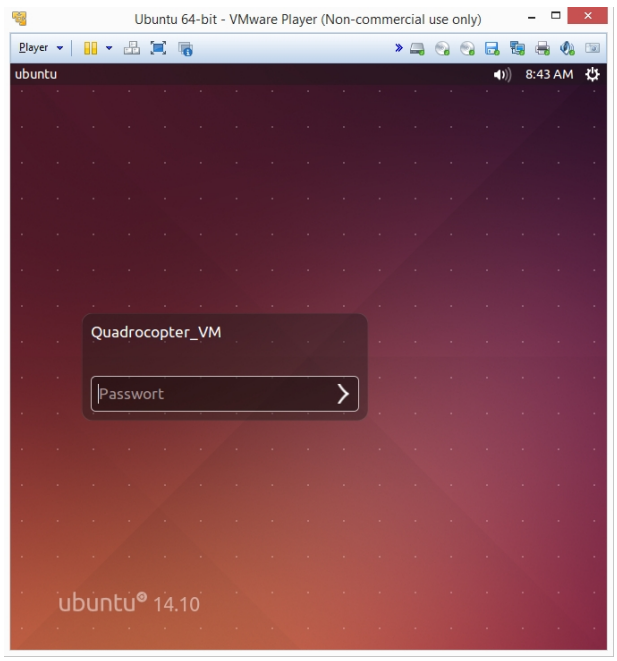


Figure 8.9: Installing Ubuntu - Step 8: Ubuntu's system login

Remark:
If the rescaling later on will not work, then after the start and login into Ubuntu, the following command needs to run in the terminal window:

```
1 sudo apt-get install open-vm-tools-desktop
```

8.2 Toolchain

A Toolchain enables compilation on the fast development computer instead on the Raspberry Pi. This is called cross compiling. To make it possible, following commands have to be called in the Ubuntu operating system which is running in the virtual machine. This description is based on the blog entry from Halherta [1].
First the cross compiling toolchain will be downloaded and set up. The following commands will be called in a terminal window.
The following command downloads the native build tools and the GIT tool which will be used to download the cross compiling toolchain.

```
sudo apt-get install build-essential git
```

Next a directory is created and then the directory is opened.

```
mkdir rpi
cd rpi
git clone git://github.com/raspberrypi/tools.git
```

The last command downloads the official cross compiling Toolchain. In the folder /home/user/rpi/tools/arm-bcm2708 are now four different folders stored which contain different toolchains.

```
arm-bcm2708-linux-gnueabi
arm-bcm2708hardfp-linux-gnueabi
gcc-linaro-arm-linux-gnueabihf-raspbian
gcc-linaro-arm-linux-gnueabihf-raspbian-x64
```

Due to the reason that the used Ubuntu is a 64 bit OS the fourth toolchain has to be used. To enable compilation also directly from the command line, the path to the Toolchain has to be added in the PATH variable of the Ubuntu system. To do so, the following actions need to be done.

```
cd ~/
nano .bashrc
export PATH=$PATH:$HOME/rpi/tools/arm-bcm2708/
    gcc-linaro-arm-linux-gnueabihf-raspbian-x64/bin
source .bashrc
```

The last command updats the PATH variable.
If all actions are successfully done, it can be proved by:

1 http://hertaville.com/2012/09/28/development-environment-raspberry-pi-cross-compiler/

```
arm-linux-gnueabihf-gcc -v
```

If all works fine the terminal window should show following output.

```
user@ubuntu:~$ arm-linux-gnueabihf-gcc -v
Using built-in specs.
COLLECT_GCC=arm-linux-gnueabihf-gcc
COLLECT_LTO_WRAPPER=/home/user/rpi/tools/arm-bcm2708/gcc-linaro-arm-linux-gnueabihf-raspbian-x64/bin/../libexec/gcc/arm-linux-gnueabihf/4.8.3/lto-wrapper
Target: arm-linux-gnueabihf
Configured with: /home/zhehe01/work/bzr/pi-build/builds/arm-linux-gnueabihf-raspbian-linux/.build/src/gcc-linaro-4.8-2014.03/configure --build=x86_64-build_unknown-linux-gnu --host=x86_64-build_unknown-linux-gnu --target=arm-linux-gnueabihf --prefix=/home/zhehe01/work/bzr/pi-build/builds/arm-linux-gnueabihf-raspbian-linux/install --with-sysroot=/home/zhehe01/work/bzr/pi-build/builds/arm-linux-gnueabihf-raspbian-linux/install/arm-linux-gnueabihf/libc --enable-languages=c,c++,fortran --disable-multilib --enable-multiarch --with-arch=armv6 --with-tune=arm1176jz-s --with-fpu=vfp --with-float=hard --with-pkgversion='crosstool-NG linaro-1.13.1+bzr2650 - Linaro GCC 2014.03' --with-bugurl=https://bugs.launchpad.net/gcc-linaro --enable-__cxa_atexit --enable-libmudflap --enable-libgomp --enable-libssp --with-gmp=/home/zhehe01/work/bzr/pi-build/builds/arm-linux-gnueabihf-raspbian-linux/.build/arm-linux-gnueabihf/build/static --with-mpfr=/home/zhehe01/work/bzr/pi-build/builds/arm-linux-gnueabihf-raspbian-linux/.build/arm-linux-gnueabihf/build/static --with-mpc=/home/zhehe01/work/bzr/pi-build/builds/arm-linux-gnueabihf-raspbian-linux/.build/arm-linux-gnueabihf/build/static --with-isl=/home/zhehe01/work/bzr/pi-build/builds/arm-linux-gnueabihf-raspbian-linux/.build/arm-linux-gnueabihf/build/static --with-cloog=/home/zhehe01/work/bzr/pi-build/builds/arm-linux-gnueabihf-raspbian-linux/.build/arm-linux-gnueabihf/build/static --with-libelf=/home/zhehe01/work/bzr/pi-build/builds/arm-linux-gnueabihf-raspbian-linux/.build/arm-linux-gnueabihf/build/static --enable-threads=posix --disable-libstdcxx-pch --enable-linker-build-id --enable-plugin --enable-gold --with-local-prefix=/home/zhehe01/work/bzr/pi-build/builds/arm-linux-gnueabihf-raspbian-linux/install/arm-linux-gnueabihf/libc --enable-c99 --enable-long-long --with-float=hard
Thread model: posix
gcc version 4.8.3 20140303 (prerelease) (crosstool-NG linaro-1.13.1+bzr2650 - Linaro GCC 2014.03)
user@ubuntu:~$
```

Figure 8.10: Prove of successfull working Toolchain

8.3 Eclipse

8.3.1 Installation of Eclipse Luna

Next, the current 64 Bit Eclipse IDE for C/C++ Developers, is downloaded from the webpage of eclipse within the Ubuntu system:

```
https://www.eclipse.org/downloads/?osType=linux
```

First the current Java runtime environment is installed.

```
sudo apt-get install openjdk-7-jre
```

Then the before downloaded eclipse is extracted and stored to /opt

```
sudo tar -xvzf eclipse-cpp-luna-SR2-linux-gtk-x86_64 -C /opt/
```

In order to access the most recent eclipse, a custom desktop link has to be created and edited.

```
sudo touch /usr/share/applications/eclipse-recent.desktop;
sudo gedit /usr/share/applications/eclipse-recent.desktop &
```

An editor will open after executing the above shown lines. Paste the following lines in order to create a valid Ubuntu desktop launcher:

```
[Desktop Entry]
Type=Application
Name=Eclipse Luna
Comment=Eclipse Integrated Development Environment
Icon=/opt/eclipse/icon.xpm
Exec=/opt/eclipse/eclipse
Terminal=false
Categories=Development;IDE;Java;
```

Optionally, a starter icon can be put to the desktop of Ubuntu by copying the above created Ubuntu desktop launcher:

```
sudo cp /usr/share/applications/eclipse-recent.desktop ~/Desktop/
```

8.3.2 Remote Debugging

Enabling remote debugging is really helpfull when a written program has not the expected behaviour. To enable remote debugging, some configurations in the Debug Configurations need to be done. This window can be found under 'Run/Debug Configurations...' In the starting window a click with the left mousebutton on 'C/C++ Remote Application' on the left part of the window needs to be done. Then click on 'New'. See figure 8.11.

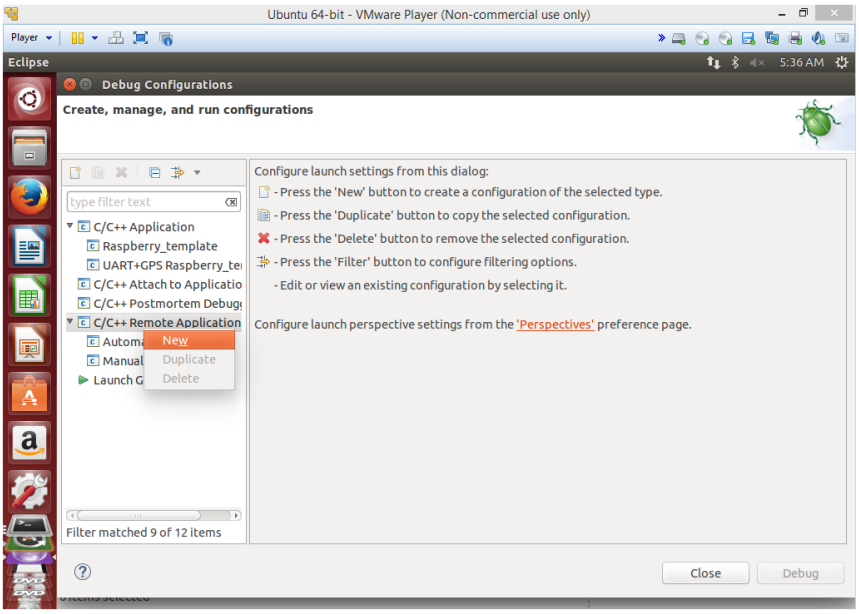


Figure 8.11: Making new Debug Configuration

In the appearing window write down first the Name 'Automatic_Debug' and then in the tab 'Main' (see figure 8.12) write down the name of the project in the textfield 'Project:' and in 'C/C++ Application:' the executable, the binary file, which is in the Debug folder of the project. Under 'Build configuration' 'Use Active' and the radio button 'Enable auto build' has to be activated. When clicking under 'Connection:' on 'New...' a new window, like in figure 8.13 appears. Here 'SSH only' and 'Next>' needs to be clicked. After that, in the next automatically popped up window all what needs to be written down can be seen in figure 8.14, and then it has to be confirmed by clicking on 'OK'. Last but not least there needs to be set a Folder which can be a existing or a new one in 'Remote Absolute File Path for C/C++ Application:' and under 'Commands to execute before application' the commands

```
1 sudo -i
2 chmod +x Raspberry_Template
```

have to be included. Last make sure that in the lower area of the window 'Using GDB (DSF) Automatic Remote Debugging Launcher' is active.

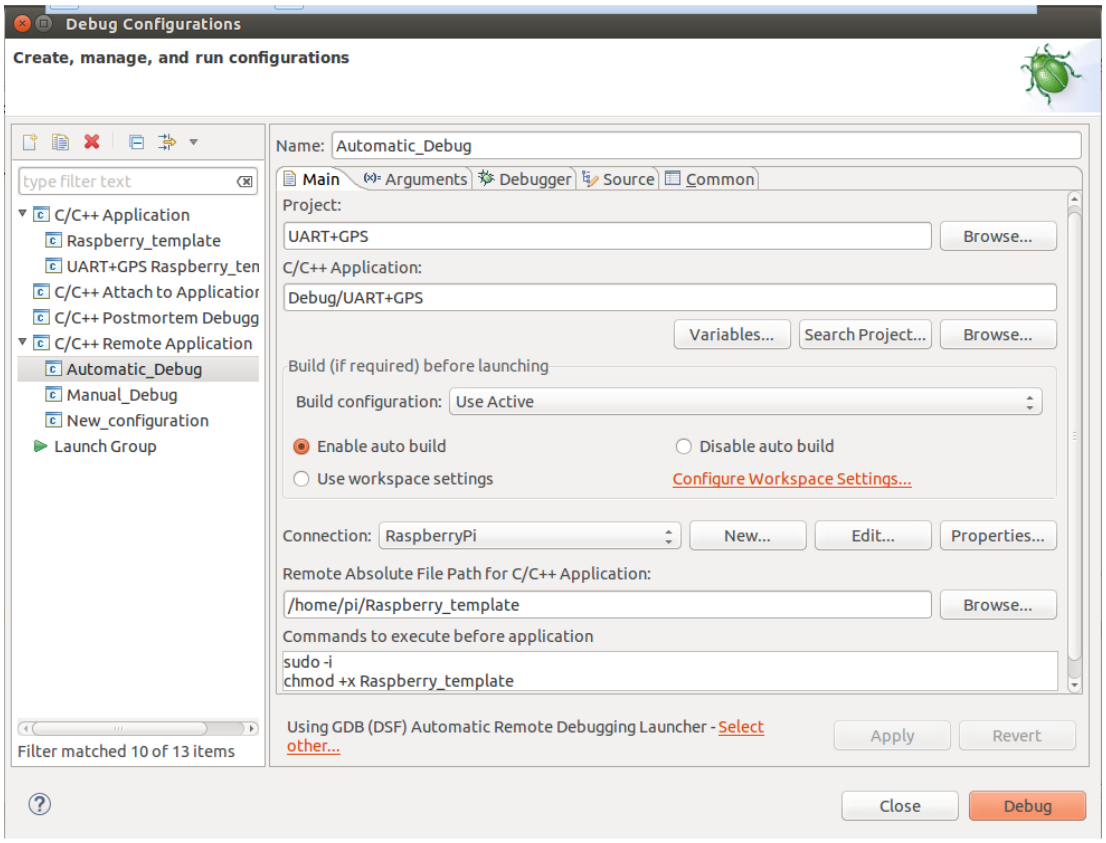


Figure 8.12: Debug Configuration Window - Main tab

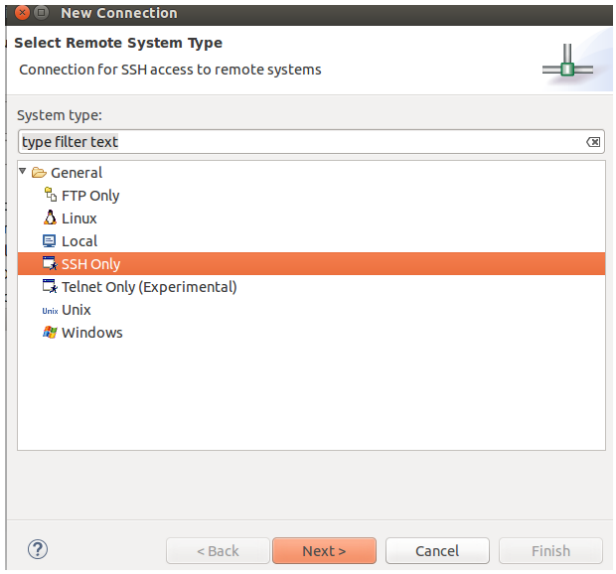


Figure 8.13: Creating new connection 1

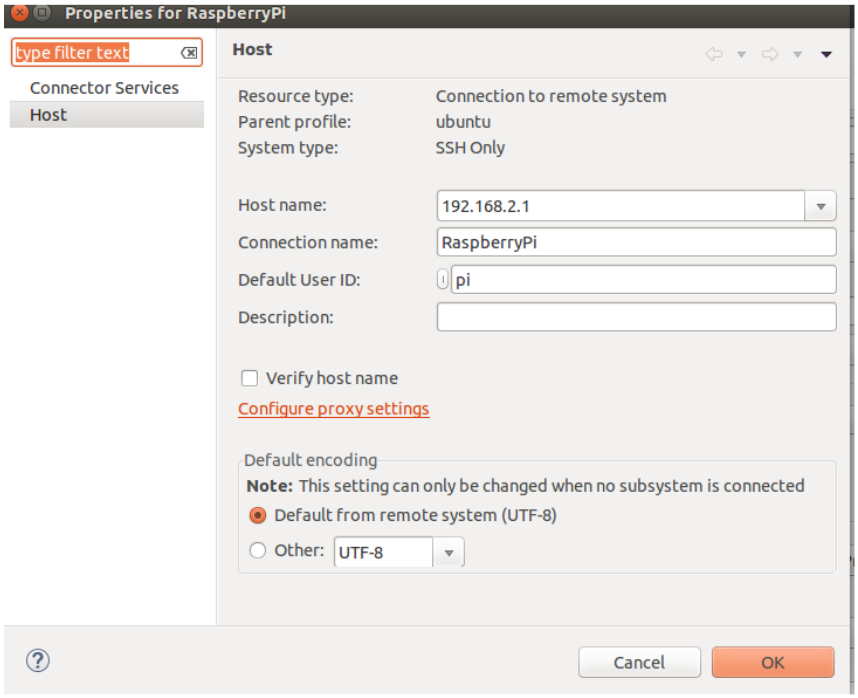


Figure 8.14: Creating new connection 2

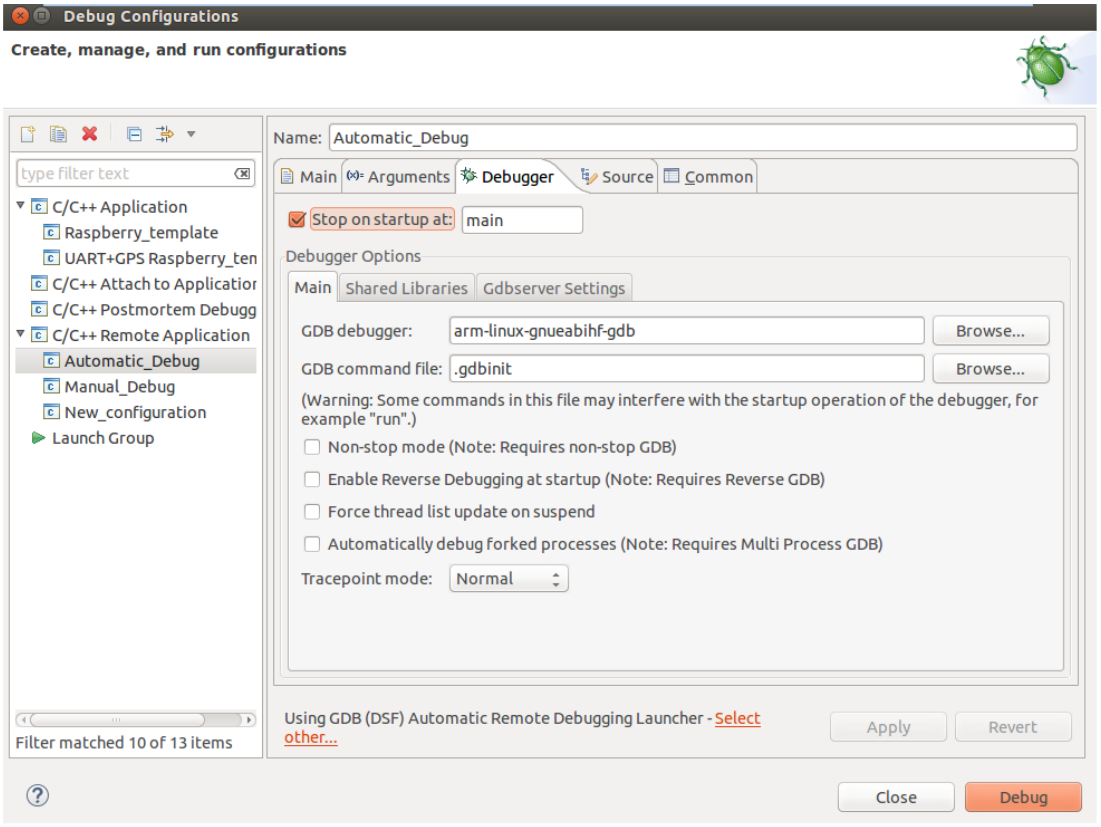


Figure 8.15: Debug Configuration Window - Debugger tab

After this is finished, in the Debugger tab, there needs in the 'GDB debugger' textfield 'arm-linux-gnueabihf-gdb' written down. This ensures that the ARM toolchain connects to the GDB server on the Raspberry (See figure 8.15).

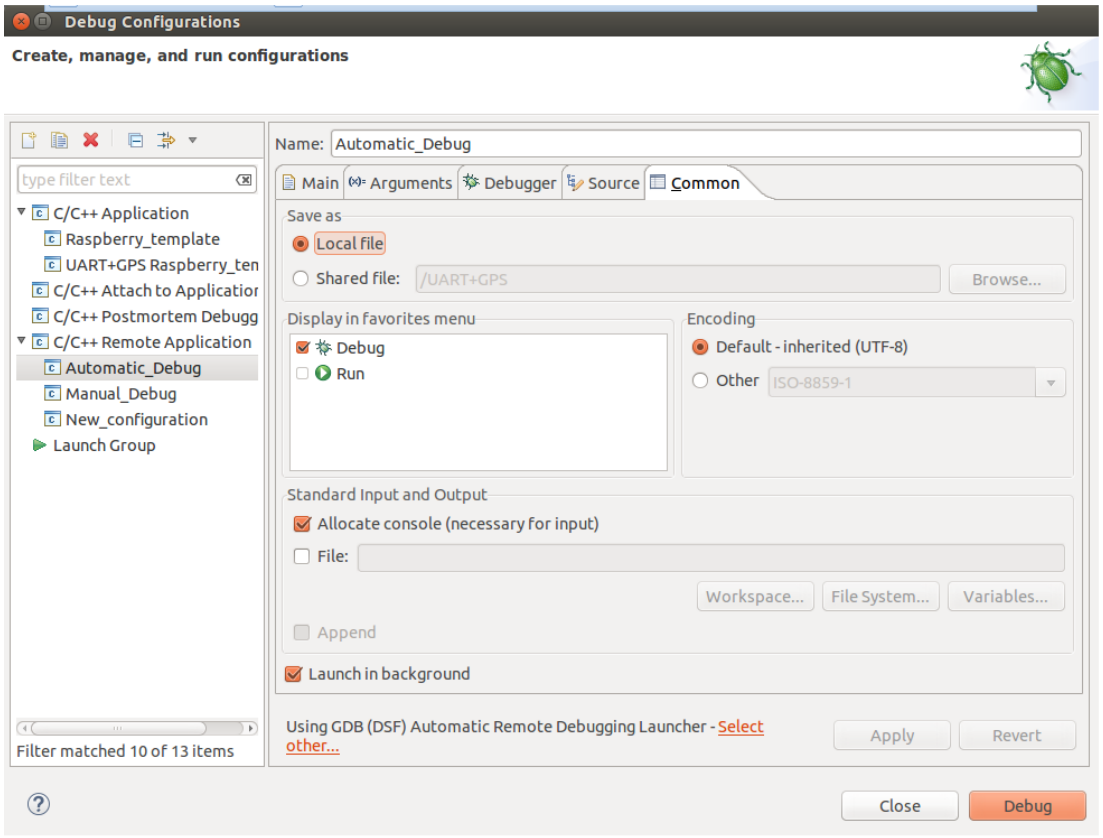


Figure 8.16: Debug Configuration Window - Common tab

Then in the 'Common' tab in 'Display in favorites menu' the 'Debug' checkbox needs to be activated. After that the window can be closed. Now all necessary actions are done and an existing project can be by debugged by clicking on the small arrow beneath the the small bug in the menu bar. Here the before saved Configuration (Automatic_Debug) has to be activated what the debugging directly starts.

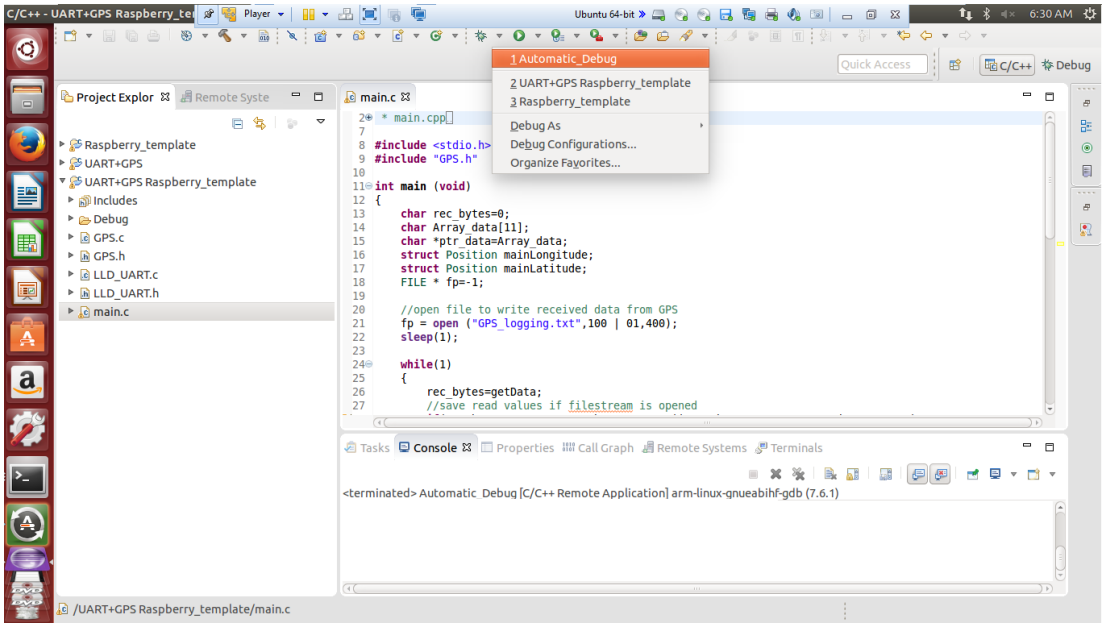


Figure 8.17: Start Debugging

9 Realtime-capable Linux Operating System

9.1 Creating a bootable SD-Card with a pre-configured Firmware

For a convenient and easy start with the Raspberry Pi, a pre-configured Raspberry Pi Firmware can be found at the SVN repository folder `/sys/RPi/`. The pre-configured Firmware Image is based on the EMLID-distribution (see also chapter 9.3). The EMLID-distribution originates from a open source project[1] for a different Raspberry Pi based Quadcopter. It comprises the RT-Kernel patch as well as several optimizations regarding the system configuration to access peripherals that fit to the needs of this project[2].

For this project, pick the archive file `EMLID_Preempt_IP192.168.2.1.zip` from the SVN repository. Unzip the file to get the extracted 8GB Firmware Image.

To write the Firmware Image to a SD Card, use the tool `Win32DiskImager`[3] for Windows based Host machines (as shown in fig. 9.1).

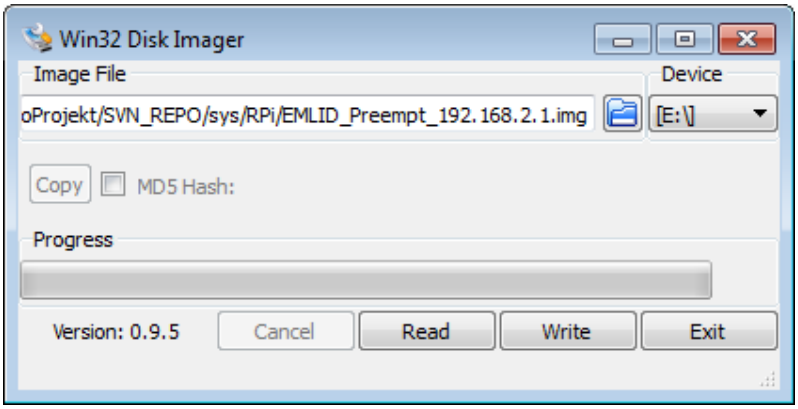


Figure 9.1: Windows tool Win32DiskImager to write a Raspberry Pi Firmware to a SD Card in a Windows Host System

1 EMLID-Distribution: http://www.emlid.com/new-raspbian-with-real-time-kernel-jan-2015/

2 The hardware setup as well as the software architecture was planned before the authors have discovered the EMLID-distribution. It was a lucky coincidence, that the EMLID-distribution met almost all core requirements of this project. Because of that, the authors could used the EMLID-distribution as a base distribution during the software development and could develop in parallel - independently from the progress of patching the Raspbian distribution to a RTOS (as shown in chapter 9.2).

3 Download `Win32DiskImager` at http://sourceforge.net/projects/win32diskimager/

For Linux based host machines (as well as the Development Environment VM), the built-in program `dd` can be used to copy the Firmware to a SD card. Therefore, the device handler `/dev/sdX` has to be determined, first! Issue the command `lsblk` and search for a disk with the same size as the SD card.
Assuming the device handler is `/dev/sdb` (use device handler of complete disk, without any numbers in the name), the Firmware can be copied with the below shown command.

Attention:
A wrong chosen device handler can destroy the operating system of the host machine!

```
cd ../path/to/svnRepository/sys/RPi/
unzip EMLID_Preempt_IP192.168.2.1.zip
sudo dd if=EMLID_Preempt_192.168.2.1.img of=/dev/sdb
```

Remark:
To use the pre-configured Firmware Image, a SD Card with at least 8GB storage capacity has to be used! Otherwise, the Firmware Image will be too big.

9.2 Building a Real-Time Linux Kernel for the Raspberry Pi

In the following chapters, the folder structure for the kernel sources and kernel developments on the Development Environment (Virtual Machine) will be evolving such that the following folder structure appears:

```
~/ ..... Home-directory of the current user
└── rpi ..... Raspberry Pi related sources and files
    ├── linux ..... Linux Kernel sources
    ├── linux-rt-rpi ..... EMLID-based Kernel sources (see chapter 9.3)
    └── tools ..... Raspberry Pi toolchain tools
```

Therefore, first create the first folder hierarchy `rpi` with the commands

```
cd ~
mkdir rpi
cd rpi
```

Next, get latest RT-Kernel patch from the official website of the Linux Kernel Organization, Inc. (`https://www.kernel.org/pub/linux/kernel/projects/rt/`). Open the link with a browser in the Development Environment VM and navigate to the current stable release of Linux kernels (version 3.18 at the time of this project). Choose the file `patch-3.18.XX-rtXX.patch.xz` and copy the URL. Use the copied link to issue the command `wget` to download the Kernel patch archive into the folder `~/rpi/linux`, similar as shown below:

```
cd ~/rpi/
mkdir linux
cd linux
wget https://www.kernel.org/pub/linux/kernel/projects/rt/3.18/patch
    -3.18.16-rt13.patch.xz
```

As the RT-Kernel patch version and the kernel source version **must match**, the next step will be to get the Linux kernel source according to the latest available RT-patch version. For that purpose, browse to `https://github.com/raspberrypi/linux/` to observe the available branches (see fig.9.2). After a matching Kernel source branch could be determined, load the Kernel sources to the Development Environment using the command `git clone [URL]`. With the additional option `-b`, the corresponding branch can be chosen:

```
cd ~/rpi/
git clone --depth=1 git://github.com/raspberrypi/linux.git -b rpi-3.18.y
```

The download of the kernel sources will take some time. According to the available bandwidth of the github servers, this may take even 1 - 2 hours!

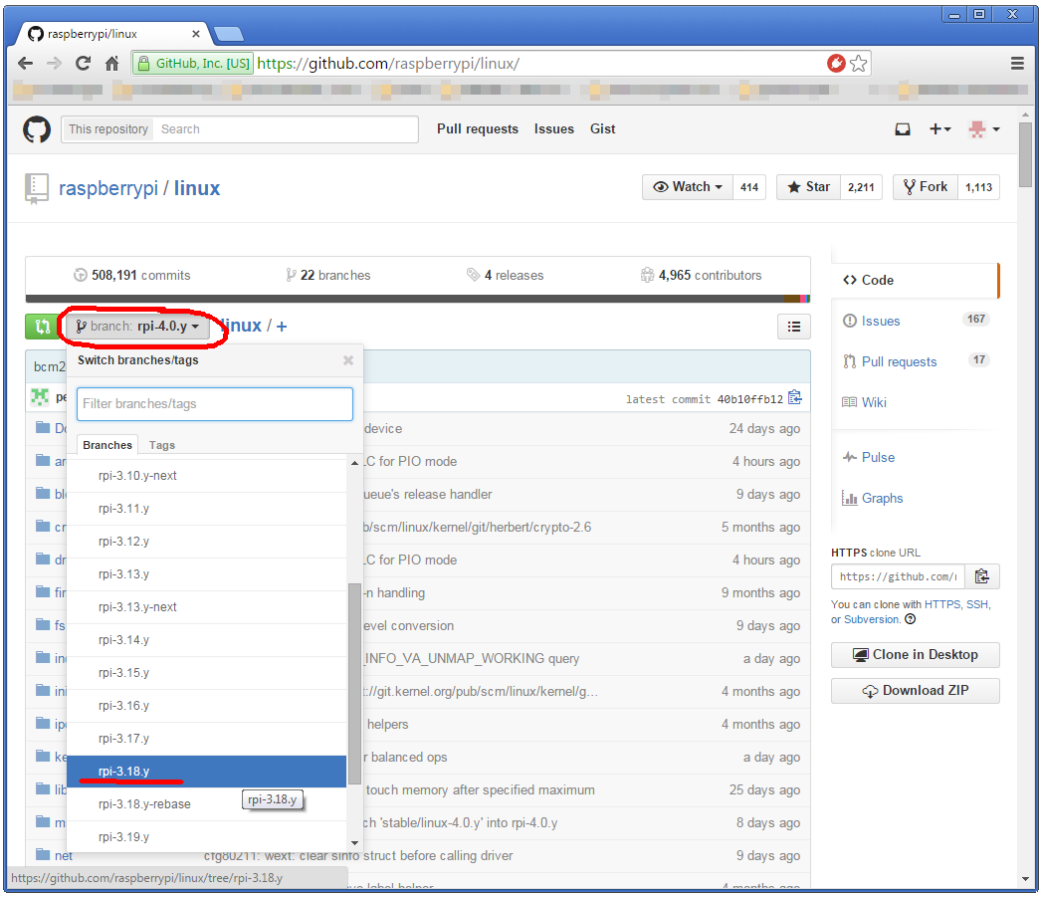


Figure 9.2: Branch list of github's repository of the Raspberry Pi's Kernel sources

Verify the downloaded kernel version by typing `make kernelversion` and check that the Kernel version number matches the Kernel version of the RT-Kernel patch.
Example:

```
user@ubuntu:~/rpi$ cd linux
user@ubuntu:~/rpi/linux$ make kernelversion
3.18.16
```

If the Kernel version numbers match, the RT-Kernel patch can be applied to the Kernel sources. First, a dry run will be executed to check for errors:

```
cd ~/rpi/linux/
xcat patch-3.18.16-rt13.patch.xz | patch -p 1 --dry-run
```

If no errors occur, the patch process can be applied without dry-run option:

```
xcat patch-3.18.16-rt13.patch.xz | patch -p 1
```

Now the Kernel sources have been patched to a Preempt_RT setup! Next, the Kernel source have to get compiled. Therefore, the Raspberry Pi cross-compile toolchain has to be downloaded, if not already given (see also chapter 8 for details):

```
cd ~/rpi/
git clone --depth=1 git://github.com/raspberrypi/tools.git
```

The following compilation process shown in this documentation is based on the article of [LMK69]. In order to cross-compile the sources in the Development Environment VM, some environment variables in the command line have to be set up:

```
export ARCH=arm
export CROSS_COMPILE=../tools/tools/arm-bcm2708/gcc-linaro-arm-linux-
    gnueabihf-raspbian-x64/bin/arm-linux-gnueabi-
export INSTALL_MOD_PATH=~/rpi/linux/outputLib
mkdir ~/rpi/linux/outputLib
```

Now, navigate to the sources and make Raspberry Pi configuration of the kernel sources.

```
cd /linux
make bcmrpi_defconfig
```

Optionally, with the command `make menuconfig`, the configuration of the Kernel source setup can be refined. But for the puposes of this project, no adaptions are required.
Next, the kernel sources can be cross-compiled, using the command below.

Remark: The option `-j` defines, how many compilation processes shall be used. Set this value to the same number as CPU cores are available at your machine to achieve the fastest possible cross-compilation.

```
make -j4
```

Attention:
The cross-compilation can take 20minutes up to 2 hours, depending on the resources and CPU power of your development machine! Nevertheless, it is far more efficient to do a cross-compilation than compiling the Kernel sources on the Raspberry Pi directly - this will take 10+ hours!

As a last step of the Kernel compilation, the standard kernel modules have to be linked and installed. According to the environment variables defined above, the kernel modules will be installed at `~/rpi/linux/outputLib/`.

```
make modules_install
```

Get the latest Raspbian distribution at official webpage of the Raspberry Pi Foundation (direct link: `http://downloads.raspberrypi.org/raspbian_latest`).
Unzip the Firmware Image and copy it to a SD card, according to the steps shown in chapter 9.1.

Boot up the device using a keyboard and a display attached to the Raspberry Pi. Log in to the system by using the credentials:

```
user: pi
password: raspberry
```

Configure the Raspbian distribution to a minimal configuration, issue the command:

```
sudo raspi-config
```

A graphical user interface will appear as shown in fig. 9.3. Navigate through the menu, to enable the following options:

```
(1) 8 Advanced Options => A4 SSH => <Enable> => <OK>
(2) 8 Advanced Options => A7 I2C => <YES> => <OK> => <YES> => <OK>
(3) 8 Advanced Options => A8 Serial => <NO> => <OK>
(4) 8 Advanced Options => A2 Hostname => <OK> => "HElikopterPi" => <OK>
(5) <FINISH> => <YES>
```

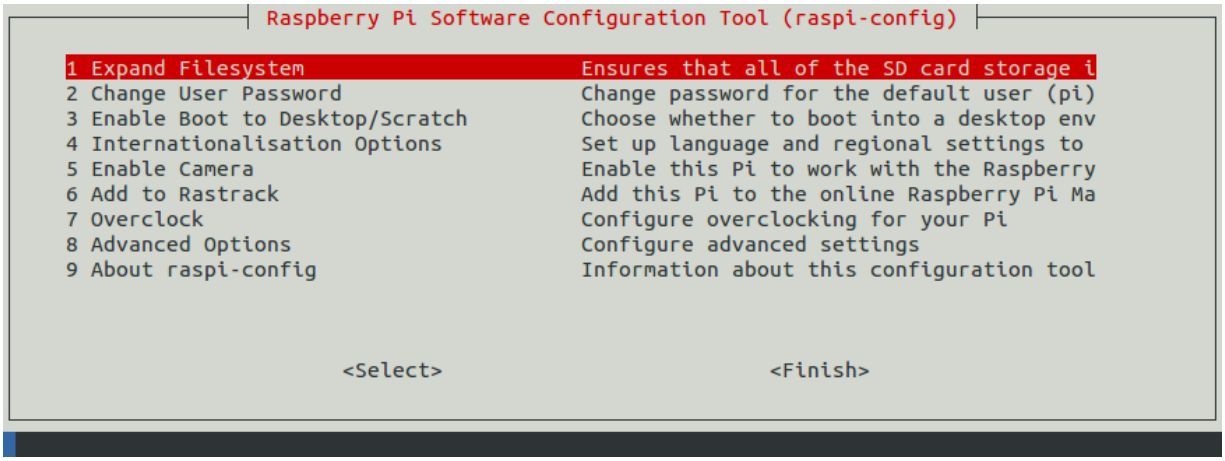


Figure 9.3: Graphical configuration tool `raspi-config` for Raspbian distribution

The Raspberry Pi will now reboot to enable all system changes. After the restart, log in again using the keyboard and display attached to the device.

Now, set the network interface to the static network address `192.168.2.1` by editing the file `/etc/network/interfaces`:

```
sudo nano /etc/network/interfaces
```

Ensure that the file is changed such that it matches the excerpt below (assuming your host machine will have the IP `192.168.2.2`):

```
#iface eth0 inet dhcp
iface eth0 inet static
        address 192.168.2.1
        netmask 255.255.255.0
        gateway 192.168.2.2
```

As a last step, the root user needs to get a password to enable a comfortable remote file transfer after the Kernel compilation:

```
pi@HElikopterPi ~ $ sudo su
root@HElikopterPi:/home/pi# passwd
Enter new UNIX password:
Retype new UNIX password:
passwd: password updated successfully
root@HElikopterPi:/home/pi# exit
exit
pi@HElikopterPi ~ $
```

Now, the Raspberry Pi can be connected with an Ethernet cable to the Host machine running the Development Environment VM. A ssh connection should be possible with the following command in the Development Machine:

```
ssh pi@192.168.22.1
```

If the ssh connection could be successfully established, log out and send kernel the compiled Kernel to the Raspbian system by typing the following commands at the Development Environment:

```
scp ~/rpi/linux/arch/arm/boot/zImage root@192.168.2.1:/boot/zImage
```

Log in at Raspberry Pi with the above state credentials and modify the boot-configuration file `/boot/config.txt` such that the line `kernel="zImage"` will be present. If this procedure is done the first time, a easy way to insert the line to the file is the command below:

```
sudo su
echo "kernel=zImage" >> /boot/config.txt
exit
```

Alternatively, use the text file editor `nano` to edit `config.txt`:

```
sudo nano config.txt
```

Last but not least, copy all kernel modules from the Development Environment VM to the Raspberry Pi using rsync as shown below. Type the following command to the terminal of the Development Environment VM:

```
sudo su
rsync -av ~/rpi/linux/outputLib/ root@192.168.2.1:/lib/
exit
```

Now the new Kernel and all corresponding Kernel modules are updated. Restart the Raspberry Pi by typing the following command at the Raspberry Pi's command line and wait until the device is up again to reconnection via ssh.

```
sudo restart
```

After you have logged in, check if the kernel update was successfully. If the new Kernel with the Preempt_RT patch has been started, the command `uname -a` should give an output similar as stated below:

```
pi@HElikopterPi ~ $ uname -a
Linux HElikopterPi 3.18.14-rt10+ #1 PREEMPT RT Mon Jun 15 21:21:16 CEST
    2015 armv6l GNU/Linux
pi@HElikopterPi ~ $
```

The relevant parts in the output are highlighted. If the RT-Kernel patch was successfully, the term `PREEMPT RT` should show up as shown above.

9.3 Pre-configured EMLID distribution for development purposes

The EMLID-distribution (http://www.emlid.com/) originates from a open source project[1] of an autonomous Raspberry Pi controlled Quadcopter, similar to this project. Different to this project, the EMLID community uses a custom hardware module to extend the hardware capabilities of the Raspberry Pi. In this project, the authors aimed and succeeded to extend the Raspberry Pi with ready-made on market available sensors and boards.

The hardware setup as well as the software architecture of this project was planned before the authors got known about the existence of the EMLID-distribution. It was a lucky coincidence, that the EMLID-distribution meets almost all core requirements of this project. Because of that fact, the authors could use the EMLID-distribution as a base distribution during the software development. In consequence, the software development could be processed in parallel to the other project tasks - especially independently from the progress of patching the Raspbian distribution to a RTOS (applying the RT-Kernel patch as shown in chapter 9.2).

Since some Kernel module driver has been developed during this project, it was also necessary to compile the custom Kernel drivers for the EMLID-distribution. Therefore, get slightly modified Kernel sources of the EMLID-distribution can be found at https://github.com/emlid/linux-rt-rpi. To download the sources to the Development Environment VM, the following commands can be used:

```
cd ~/rpi/
git clone --depth=1 git://github.com/emlid/linux-rt-rpi.git
```

This will download the EMLID Kernel sources that are already modified with an appropriate RT-Kernel patch. All source files will be stored to the directory `~/rpi/linux-rt-rpi/`.

In order to compile the EMLID kernel sources, the procedure is identical as shown in chapter 9.2 except application of the RT-Kernel patch.

1 Download at: http://www.emlid.com/new-raspbian-with-real-time-kernel-jan-2015/

To download the complete EMLID-distribution Firmware, see http://www.emlid.com/new-raspbian-with-real-time-kernel-jan-2015/ or use the pre-configured Firmware of the projects SVN repository at `/sys/Rpi/`.

9.4 Configurations of peripherals and busses

Due to the specific hardware extension boards and sensors used in this project (see also chapter 5.1.1), some minor and basic configurations are needed for a flawless operation of the Raspberry Pi platform.

The following aspects have to be met:

- **UART** port has to be unused (not utilized as kernel debug output) since the GPS receiver of the GPS Hat extension board will occupy the UART channel.
- **I2C** bus (Bus No. 1, Bus No. 0 is not used in this part of the project) has to be set to a baudrate of 400kBit/s in order to meet the busload requirements.
- **Network** configuration has to be set up (static IP address for the most common development scenario: Raspberry Pi attached via wire to the Development Machine).

Network configuration

Set the network interface to the static network address `192.168.2.1` by editing the file `/etc/network/interfaces`:

```
sudo nano /etc/network/interfaces
```

Ensure that the file is changed such that it matches the excerpt below (assuming your host machine will have the IP `192.168.2.2`):

```
#iface eth0 inet dhcp
iface eth0 inet static
        address 192.168.2.1
        netmask 255.255.255.0
        gateway 192.168.2.2
```

Perform the command shown below, to update the network configuration of the Linux Operating System. After the update, the changed network configuration will take place:

```
sudo /etc/init.d/networking restart
```

9.4.1 Raspbian distribution specific

To configure the Raspbian distribution such that the **UART** port can be used by the additional peripherals (GPS receiver) and the **I2C bus** is activated, issue the following command:

```
sudo raspi-config
```

A graphical user interface will appear as shown in fig. 9.4. Navigate through the menu, to enable the following options:

```
(1) 8 Advanced Options => A7 I2C => <YES> => <OK> => <YES> => <OK>
(2) 8 Advanced Options => A8 Serial => <NO> => <OK>
(3) <FINISH> => <NO>
```

Prevent the Raspberry Pi from rebooting, since the required data rate of the I2C bus has to be configured first.

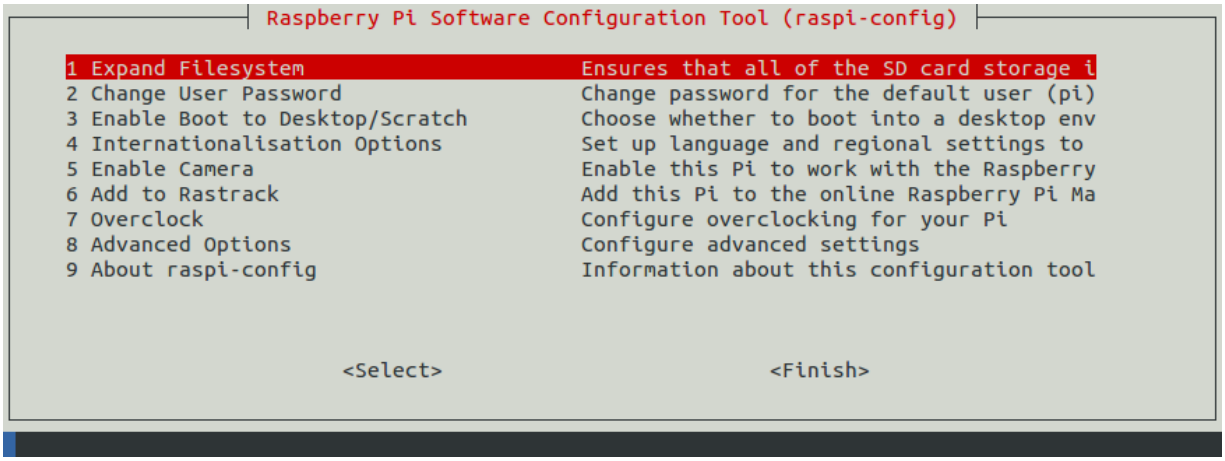


Figure 9.4: Graphical configuration tool `raspi-config` for Raspbian distribution

To configure the required **400kBit/s bauddrate**, navigate to the folder `/etc/modprobe.d/`. Unlike for the EMLID-distribution, file kernel module configuration file `i2c.conf` does not exist. To create the file with the proper setup of the I2C bus, perform the following command:

```
sudo bash -c "`echo 'options i2c_bcm2708 baudrate=400000' > /etc/modprobe.
    d/i2c.conf"
```

Now reboot the Raspberry Pi so that the configuration changes can take place.

9.4.2 EMLID distribution specific

For the EMLID-distribution, the UART port is configured to the required setup by default. No Kernel debugging messages are put to the serial port. But the default I2C baudrate is set to 1Mhz which is to much. The highest common frequency of the sensors participating on the I2C bus is 400kHz.

I2C configuration

To specify the required datarate of the I2C bus, navigate to the folder `/etc/modprobe.d/` and edit the file `i2c.conf`. Ensure that the content of the file looks like shown below:

```
1 options i2c_bcm2708 baudrate=400000
```

Therefore, open `nano` and edit the file `i2c.conf`:

```
1 sudo nano /etc/modprobe.d/i2c.conf
```

10 Software structure

10.1 Software layer and structure concept

One of the core goals of the layering concept is the maximization of code re-usability. Since major parts of the software is intended to run on multiple platforms, it is mandatory to keep a strict separation between hardware-dependent and hardware-independent software.

Furthermore, since the hardware is highly modularized, the software shall reflect this modularity as close as possible to ease the interchangeability of sensors. In detail, the modularity encompasses the microprocessor platform (in this project: `Raspberry Pi B+`) and several extension boards equipped with sensors. In table 10.1, a detailed comparison between hardware modularity and software layering is shown.

To achieve the goal of a maximal code re-usability, the software shall be structured in four general layers (see fig.10.1). Within each ***Layer***, several ***Functional Units*** are defined in order to divide the software into logically separable modules.

1. **Application Layer (app)**
   This layer contains all high-level software that is necessary for the control of the quadrocopter. Control loops, position hold control, autonomous landing control and supervising functions are here located.

2. **Signal Processing Layer (sig)**
   In order to give a flexible framework for filtering and data fusion, a dedicated layer is introduced. The raw sensor data shall be filter (e.g. lowpass filtering). In a second step, the received data shall be fusioned in order to achieve an robust and reliable orientation representation of the quadrocopter.

3. **Hardware Abstraction Layer (hal)**
   Since all software of the `Application Layer (app)` and `Signal Processing Layer (sig)` shall be system independent, the `Hardware Abstraction Layer` provides all drivers for the used sensors and extension boards. The interface towards the `Signal Processing Layer (sig)` will generalize the data flow, independent of the used sensors.
   The interfaces towards the `Low-Level Driver Layer (LLD)` will be called **Low-Level Driver Interfaces (LLD_IF)**. These interfaces shall abstract the access the low-level drivers that are usually microprocessor-specific to the used hardware platform (here: Raspberry Pi B+).

4. **Low-Level Driver Layer (LLD)**
   The `Low-Level Driver Layer` contains all drivers needed for low-level data communication like `UART`, `I2C` or `SPI`. For the scope of this project (MasterQuad 2015), all needed low-lever drivers are already provided by the chosen operating system.

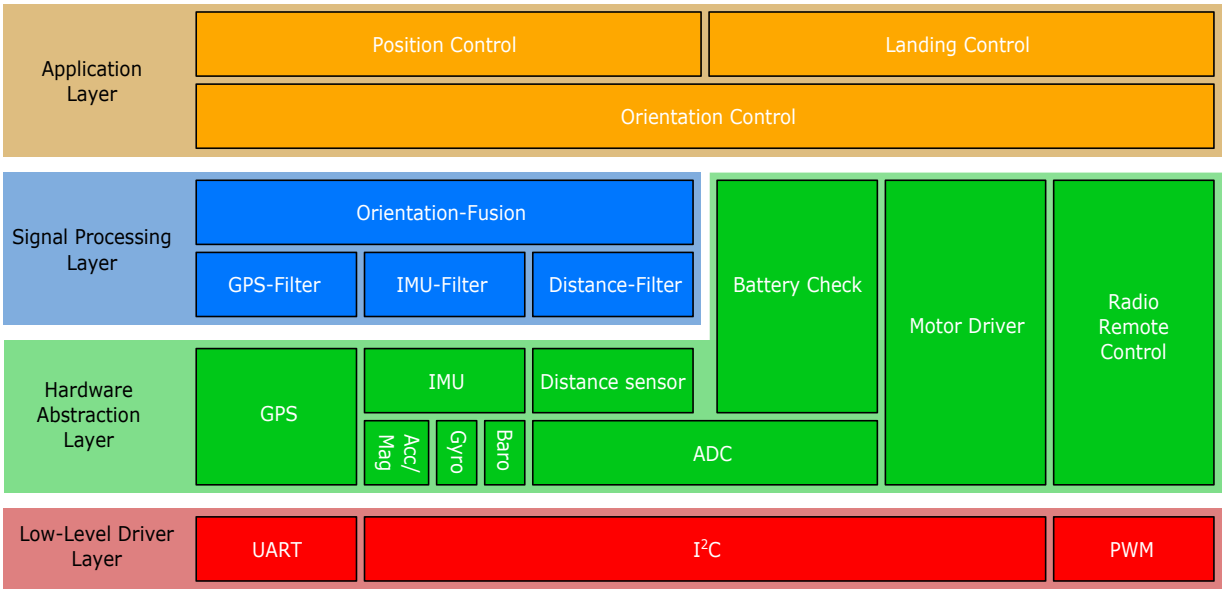


Figure 10.1: Software layers with functional units of project MasterQuad 2015

Software layer	**Equivalent interchangeable part**
Application Layer (APP)	Control loops and supervising/monitoring functions that are usually independent of the used hardware. ***Example:*** *GPS-Position Hold functionality*
Signal Processing (SIG)	**Signal conditioning** for used sensors. Additionally for sensor fusion (partly sensor-specific: parameterization of fusion algorithms are usually dependent on the signal/noise properties of sensors). ***Example:*** *Kalman-Filter (sensor fusion)*
Hardware Abstraction Layer (HAL)	**Drivers** for extension boards equipped with sensors for measurements of e.g. acceleration, gyro and distance-to-ground. ***Example:*** *GPS-Sensor (extension board)*
Low-Level Driver Layer (LLD)	Mircoprocessor platform incl. Operating System (if present) and drivers for bus communication (I^2C, etc). ***Example:*** *Raspberry Pi B+ Board*

Table 10.1: Comparison between software layers and hardware modularity

10.2 Overview on the functional units

In this section, all functional units as depicted in fig. 10.1 are described to get a comprehensive overview on the software architecture.

10.2.1 Application Layer

- **Orientation control**
  This functional unit comprises the control loop for the rotational stabilization along the XYZ-axis of the Quadrocopter.
- **Position control**
  This functional unit realizes the control for the translational stabilization along the XYZ-axis of the Quadrocopter.
- **Landing control**
  This functional unit realizes the control for a landing procedure of the Quadrocopter.

  Important remark:
  The implementation of the functional unit `Landing control` *has* ***NOT*** *been part of the authors of this project work.*

Unfortunately, none of the above mentioned control loops have been implemented during the processing period of this project work.

10.2.2 Signal Processing Layer

- **Orientation fusion**
  This functional unit comprises the complete implementations of a Complementary Filter and a Kalman Filter for the sensor fusion of sensor signals of the IMU. Additionally, a generic library for mathematical matrix operations including matrix inversion has been implemented here.

  In the implementation of the authors of this project work, the acceleration sensor, magnetic field sensor and the gyroscope sensor are used to build a comprehensive orientation representation of the Quadrocopter (see chapter **??**).
- **GPS-Filter**
  This functional unit realizes an optional filter to post-process the raw GPS-Data received by the GPS receiver.
- **IMU-Filter**
  This functional unit realizes an optional filter bank to post-process the raw sensor data provided by the sensors of the Inertial Measurement Unit (IMU).

- **Distance-Filter**
  This functional unit features an optional post-processing filter of the raw sensor data of the distance-to-ground sensor(s).

  Important remark:
  The implementation of the functional unit `Distance-Filter` *has* ***NOT*** *been part of the authors of this project work.*

10.2.3 Hardware Abstraction Layer

- **GPS**
  This functional unit comprises the parsing functionality for the GPS data. The absolute 3D position is evaluated and made accessible to the Signal Processing Layer.
- **IMU**
  The functional unit `IMU` bundles all sensors built-in on the IMU sensor board. This unit provides a consistent state representation of all inertial measurements of the Quadrocopter. The following functional units are providing raw sensor data:
  - **Acc/Mag**
    This functional unit delivers the raw measurement data of the acceleration data (translational acceleration) and the magnetic field strength components for each of the three axis in space. It comprises the necessary communication procedure to acquire new measurement data over I2C. Additionally, a conversion of the raw values to a SI-unit representation is performed.
  - **Gyro**
    This functional unit delivers the raw measurement data of the gyroscope data (rotational speed) along each of the three axis in space. It comprises the necessary communication procedure to acquire new measurement data over I2C. Additionally, a conversion of the raw values to a SI-unit representation is performed.
  - **Baro**
    This functional unit delivers the raw measurement data of the atmospheric air pressure. It comprises the necessary communication procedure to acquire new measurement data over I2C. Additionally, a conversion of the raw values to a SI-unit representation is performed.
- **Distance sensor**
  This functional unit provides access to the distance to ground sensor over I2C.

  Important remark:
  The implementation of the functional unit `Distance sensor` *has* ***NOT*** *been part of the authors of this project work.*

- **ADC**
  This functional unit features the access interface to the A/D converter over I2C. For each of the 4 channels of the A7D converter, a conversion can be triggered and the corresponding value be read.

  Important remark:
  The implementation of the functional unit `ADC` *has* ***NOT*** *been part of the authors of this project work.*

- **Battery Check**
  This functional unit utilizes the functional unit `ADC` to read and supervise the voltage level of the battery of the Quadrocopter.

- **Motor Driver**
  This functional unit provides an interface to write speed levels the 4 (up to 8) motor drivers of the Quadrocopter over I2C.

- **Radio Remote Control**
  This functional unit realizes an interface to the Custom Kernel Driver to read the Graupner PPM signal (see chapter 13). Additionally, depending on the state of the Quadrocopter and input signals of the remote control, the Quadrocopter is set to an autonomous flight or autonomous landing mode.

  Important Remark:
  The functional unit `Radio Remote Control` has **NOT** been implemented during the processing period of this project work.

10.2.4 Low-Level Driver Layer

- **UART**
  The low-level driver to access the UART functions of the Raspberry Pi's System on Chip (SoC) where delivered ready-made by the chosen Linux Operating.

- **I2C**
  The low-level driver to access the I2C functions of the Raspberry Pi's System on Chip (SoC) where delivered ready-made by the chosen Linux Operating.

- **PWM**
  This functional unit comprises the access to the Graupner PPM signal. Therefore, a Custom Kernel Driver has been developed to precisely measure the timing of the PPM sum signal directly from the GPIO pins of the Raspberry Pi board (see chapter 13.2).

11 UDP-based network connection to MATLAB

11.1 C-Library for UDP-based network access

To assist and ease the development of the sensor fusion of the IMU data (see chapter **??**), a dedicated C-Library for a UDP-based network connection to MATLAB has been implemented. This way, raw and fusioned data can be displayed and processed in a convenient way by using MATLAB/Simulink.

The library `udpLib` is also capable to enrich all data packets with a precise timestamp according to the `POSIX.1b` specification. This specification defines a structure composed of a two `long int` values, as shown in listing 11.1. The first value stores the elapsed seconds since `Epoch` (Unix timestamp, seconds elapsed since January 1st, 1970). The second value stores the nanoseconds elapsed since the start of the current second.

```
1 struct timespec
2 {
3   long int tv_sec;     /* Seconds since Epoch.  */
4   long int tv_nsec;    /* Nanoseconds.  */
5 };
```

Listing 11.1: C-Code snippet of a POSIX.1b conform time specification structure

Important remark: At the current state, there is **no** automatic time synchronization between local and remote machine implemented in the library `udpLib`!

To acquire the precise timestamps, the external library `librt` has to be linked to the Eclipse Project that utilizes `udpLib`. Otherwise, a linking error will occur during compilation time.

11.1.1 Adding librt to the cross-compiler's linker

To add `librt` in Eclipse, navigate in the menu to `Project -> Properties`. A new window will appear as depicted in fig. 11.1. At the very left, select `C/C++ Build` and expand the menu to click on `Settings`. In the window on the right, there will show up a tabular menu. Select the tab `Tool Settings` and navigate to `Cross G++ Linker -> Libraries`.

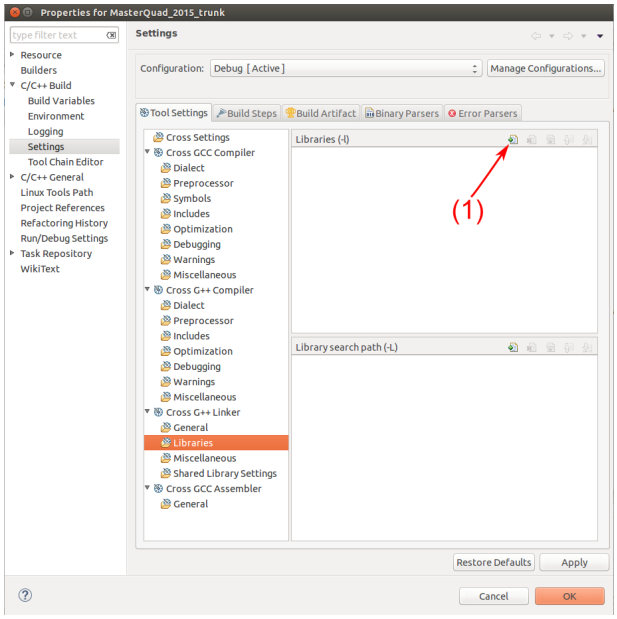


Figure 11.1: Settings menu window of Eclipse to add the library `librt` to the cross-compilers linker

In the right, a two-folded textbox with the heading `Libraries (-l)` and `Library search path (-L)` will appear. Click on the Plus-Icon of the upper textbox (`Libraries (-l)`) as indicates with the red arrow in fig. 11.1.

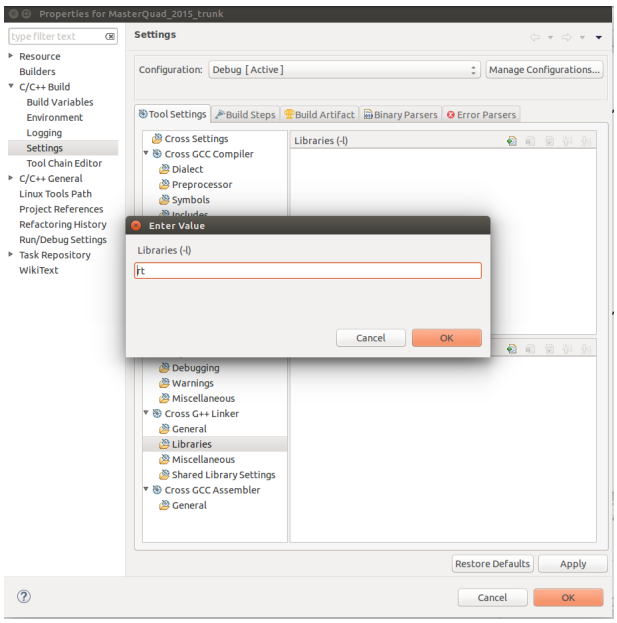


Figure 11.2: Adding the library `librt` to the cross-compilers linker

In consequence, a input window will appear as shown in fig. 11.2. Type '`rt`' into the

input field and press 'Ok'. Now the textbox `Libraries (-l)` has a new entry called `rt` (compare with fig. 11.3). Eclipse is now ready to compile and link `udpLib` successfully. Finally, close the settings window by pressing the Ok-Button.

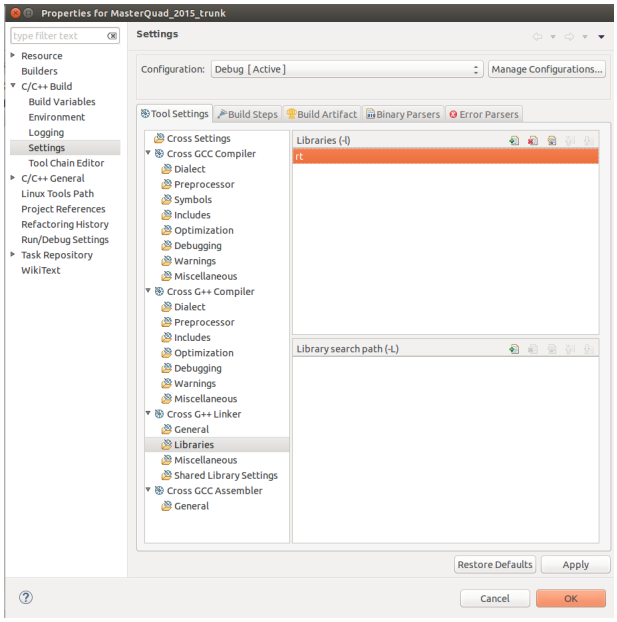


Figure 11.3: List of linked libraries of the cross-compiler in Eclipse

For the pre-configured Eclipse Project delivered in the Development Environment of this project the library `librt` is already correctly linked.

11.1.2 Basic network library

The network library for this project can be found in `/impl/trunk/matlab`. The basic functionality is implemented in `udpLib.h` and `udpLib.c`, respectively. With `udpLib`, a UPD-connection can be established and timestamped data packets sent and received.

It is assumed that on both ends of the connection, this library is utilized. Otherwise, it has to be guaranteed, that a symmetric UDP-connection is established. That means, that both machines (local and remote) are using the same port to receive data. This is visulized in fig. 11.4.

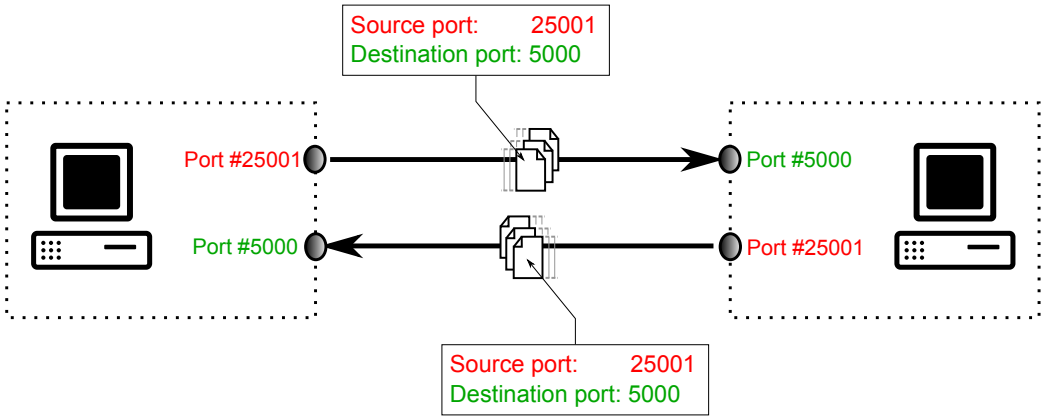


Figure 11.4: Symmetric UDP connection setup of udpLib. Both machines have the same port numbers for incoming and outgoing traffic opened. In consequence, to reply on an incoming UDP-packet it is sufficient to swap the source and destination IP-address. The port numbers will not change. With this swapped address data, the communication partner gets addressed.

To establish a network connection, the function `g_halMatlab_initConnection_i32()` has to be called. A IPv4 network address and a destination port has to be given as function parameters. The source port will be chosen automatically. After the connection has been established, data packets can be sent and received by calling `g_halMatlab_sendPacket_bl()` (without timestamps), respectively `g_halMatlab_recvPacket_ui32()`. To send packets that are enriched by precise timestamps, the function `g_halMatlab_sendRtPacket_bl()` can be called. For further details on the stated functions, please consult the code documentation for this project (see also appendix **??**).

To get a simple but comprehensive exemplary code snippet for the usage of the `udpLib`, the test files (see SVN folder `/impl/trunk/matlab/tst/`) can be studied. All crucial steps and functions are used there.

To ease the communication with specific data types (raw IMU data as well as data delivered by the sensor fusion), there has been implemented two extensions on the `udpLib`, shown in chapter 11.1.3 and chapter 11.1.4 below.

11.1.3 IMU specific network library

To simplify a communication specifically to transmit the raw IMU data from the Raspberry Pi Platform to a MATLAB/Simulink Model, the extended library `udpImuLib` has been created. Identical to basic `udpLib`, first of all a UDP-connection has to be established to use the sending and receiving functions. The function `g_halMatlab_initConnection_i32()` of the library `udpLib` has to be called.

After a connection has been established, the functions `g_halMatlab_sendImuState_bl()` and `g_halMatlab_recvImuState_bl()` can be accessed to send and receive IMU data,

marked with a precise timestamp. Different than in the basic `udpLib`, here shall a structured data variable be given as function parameter. This ensures the order and size of payload data in the UDP-packet and simplifies the parsing of the received data.

The packet data structure can be expanded as shown in listing 11.2 below.

```
typedef struct{
  /* Timestamp */
  struct
  {
    long int tv_sec;     /* Seconds since Epoch.  */
    long int tv_nsec;    /* Nanoseconds.  */
  } timestamp_st;

  /* Raw IMU data */
  struct{

    /* Acceleration data (X,Y,Z) */
    struct{
      double  x_f64;  //!< x-component of a 3D vector data
      double  y_f64;  //!< y-component of a 3D vector data
      double  z_f64;  //!< z-component of a 3D vector data
    } acc;

    /* Magnetometer data (X,Y,Z) */
    struct{
      double  x_f64;  //!< x-component of a 3D vector data
      double  y_f64;  //!< y-component of a 3D vector data
      double  z_f64;  //!< z-component of a 3D vector data
    } mag;

    /* Gyroscope data (yaw,pitch,roll) */
    struct
    {
      double l_yaw_f64;
      double l_pitch_f64;
      double l_roll_f64;
    } gyro;

    double temperature_f64;
    double pressure_f64;

  } imuState_st;

} halMatlab_rtImuPayload;
```

Listing 11.2: C-Code snippet of the expanded IMU data packet structure

11.1.4 Signal Layer specific network library

An additional extension to the `udpLib` has been implemented, to ease the transmission of sensor fusioned orientation data. For test and validation purposes, the output of the Complementary-Filter and Kalman-Filter has been observed via MATLAB/Simulink (see chapter **??**). The extended library `udpSigLib` eases therefore the network access.

The below shown listing 11.3 shows the expanded data packet structure to transmit orientation data from the Raspberry Pi to a remote machine, and vice versa.

```
typedef struct{
  /* Timestamp */
  struct
  {
    long int tv_sec;     /* Seconds since Epoch.  */
    long int tv_nsec;    /* Nanoseconds.  */
  } timestamp_st;

  /* Orientation data in space (roll, pitch, yaw) */
  struct{
    double roll_f64;
    double pitch_f64;
    double yaw_f64;
  } sigState_st;

} halMatlab_rtSigPayload;
```

Listing 11.3: C-Code snippet of the expanded Orientation data packet structure

Identical to basic `udpLib`, first of all a UDP-connection has to be established to use the sending and receiving functions. The function `g_halMatlab_initConnection_i32()` of the library `udpLib` has to be called.

After a connection has been established, the functions `g_halMatlab_sendSigState_bl()` and `g_halMatlab_recvSigState_bl()` can be accessed to send and receive IMU data, marked with a precise timestamp. Different than in the basic `udpLib`, here shall a structured data variable be given as function parameter as depicted in listing 11.3. This ensures the order and size of payload data in the UDP-packet and simplifies the parsing of the received data.

Another data structure called `halMatlab_rtSigAllStatePayload` is defined in `udpSigLib` that comprises a precise timestamp, complete raw IMU data and two times the orientation data of the quadrocopter (one output of the Complementary-Filter, another output of the Kalman-Filter). Nevertheless, this data packet structure is considered for test and validation purposes only. It should not be used in an operational (actual flying) code, due to the huge payload of the data packets and the resulting network traffic.

11.2 MATLAB/Simulink S-Function

As a counterpart to the C library `udpLib`, a special block for MATLAB/Simulink is delivered to handle a UDP-connection according to `udpLib`. To access the network peripherals out of MATLAB/Simulink, a S-Function had to be created that also utilizes the C library `udpLib` (as shown in chapter 11.1). The complete S-Function block acts as a signal source and provides the received UDP-data packets of a `udpLib`-connection as time-discrete signals in the Simulink model.

Nevertheless, as the `udpLib` provides functions to send and receive UDP-Packets, it would also be possible to implement a sink or a block with inputs and outputs to loop data through MATLAB/Simulink. Since during this project the MATLAB/Simulink model has been used for validation purposes only, such complex setups has been not regarded yet.

11.2.1 Block creation with S-Function builder

To create a Simulink model with a network connection functionality, a S-Function template has to be created, firstly. Together with the template block, a C-Code template will be created that has to be extended by`udpLib`-function calls. Last but not least, the received data packets have to be parsed and translated into Simulink signals.

As an step-by-step example, a Simulink block to read all raw IMU values via network from the Raspberry Pi shall be created in this section (used version: `MATLAB R2013a`).

To create easily a S-Function block, the S-Function Builder of MATLAB/Simulink can be used as shown below. First, start MATLAB/Simulink and create a new blank model. Open the Simulink Library Broswer and click on `Simulink -> User-Defined Functions`, as depicted in fig. 11.5 (see marked area '1.').

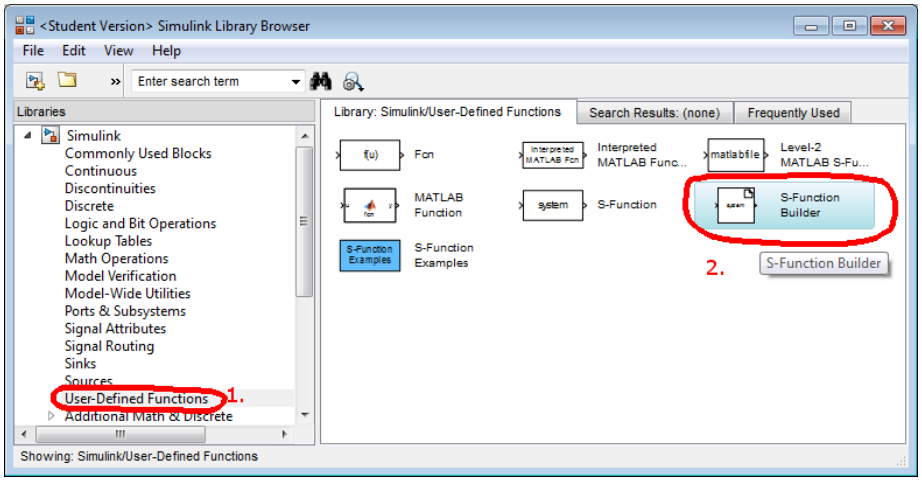


Figure 11.5: S-Function Builder 01

Drag the icon S-Function Builder into your model, such that your model will look similar to fig. 11.6. After you have placed your S-Function Builder block properly, perform a double-click on the newly created Simulink block.

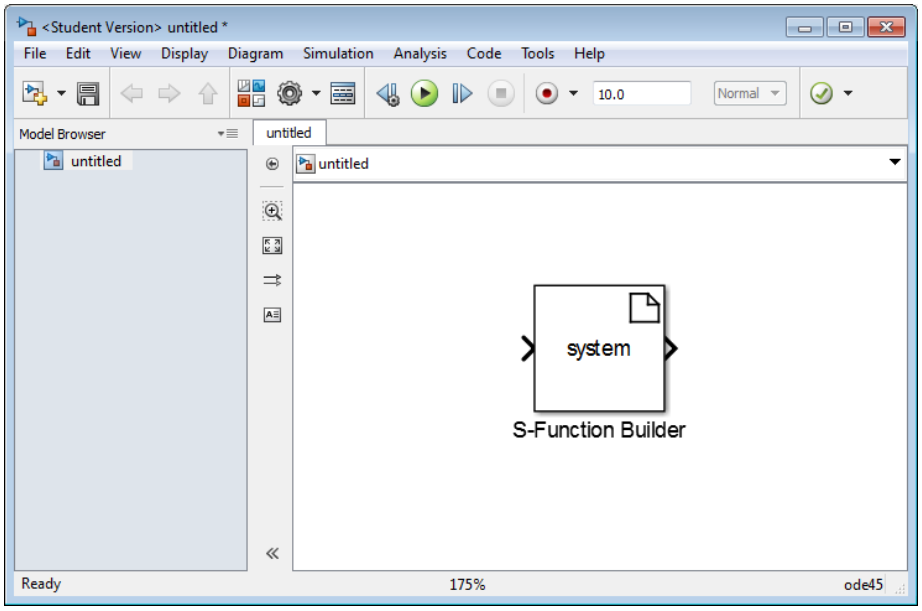


Figure 11.6: S-Function Builder 02 (Simulink model)

A window will pop up, as depicted in fig. 11.7. Choose an appropriate S-Function name (in this example '`myUdpSource`' is used) and type it in the textbox in top area of the configuration window. Next click on `Initialization` of the tab menu. You will see four properties `S-function settings` in the middle of the window. In case you are creating a signal source, as shown in this document, ensure that all state numbers are set to zero!

Remark:
Since MATLAB/Simulink treats all Simulink blocks as state space systems, internally, a simple signal source has zero states. The received UDP-packets are simply translated into Simulink signals without further processing. If you want to model a specific system behavior with your S-Function block, you might have some states (usually discrete states, since the UDP-packets will arrive in a time-discrete and isochronous manner).

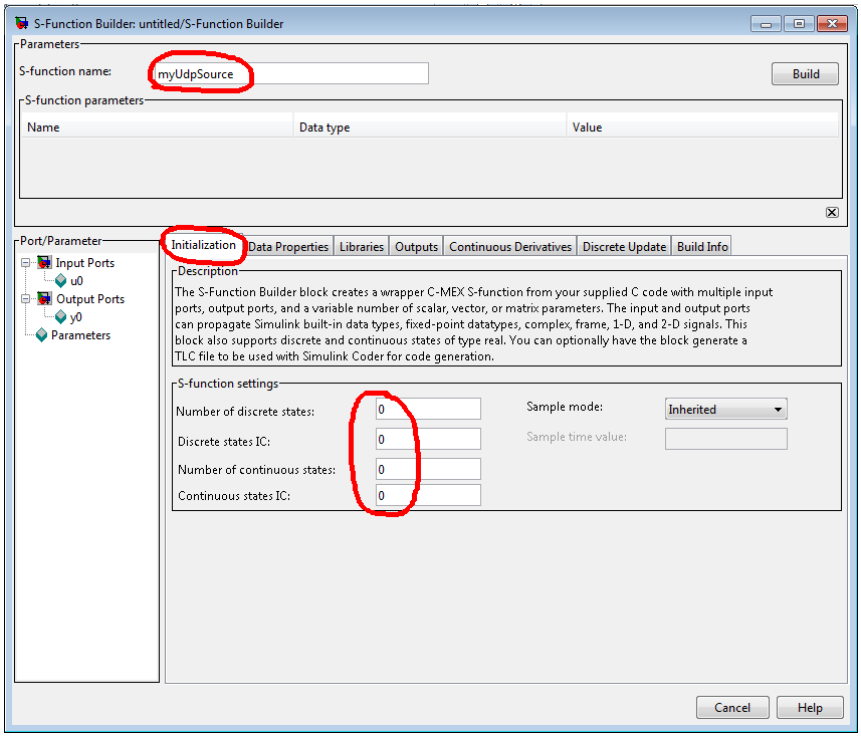


Figure 11.7: S-Function Builder 03

Since a signal source shall be created here, the S-Function block does not need to have any inputs. In consequence, all input ports have to be deleted. To edit the signal ports of the S-Function block, click on `Data Properties` of the tab menu (see fig. 11.8, mark '1.'). In the middle of the window, a second tab menu called `Port and Parameter properties` will appear. Select the tab `Input ports` and delete the default entry.

To delete the default entry, select the entry by clicking on the corresponding row (see fig. 11.8, mark '2.'). Press the red X-Button to finally delete the entry, as shown in fig. 11.8 (mark '3.').

Now, the output ports of the S-Function block can be defined. Click on the tab `Output ports`. In case there are any default entries, delete all.

In this example all raw IMU values shall be read via network and fed into the Simulink model. For this purpose, we create for each sensor one output port. It is important to consider that some of the sensors such as accelerometer or gyrosensor will have measurement values for each axis in space (X, Y and Z, respectively roll, pitch and yaw). In consequence, some output ports will have a vector as output type. This will be represented in the `Rows` value as depicted in fig. 11.9 (see mark '2.')

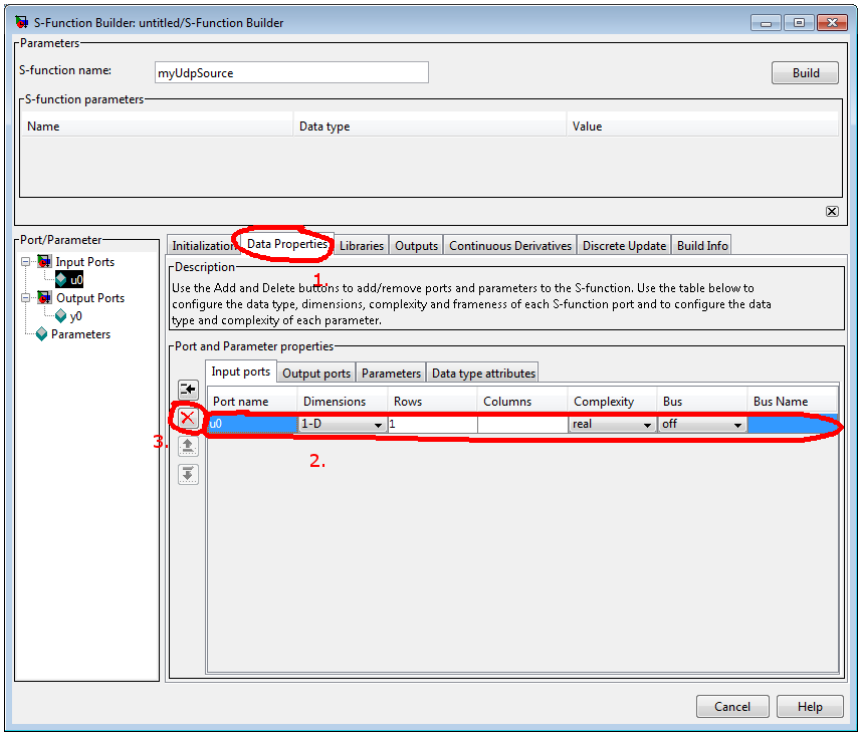


Figure 11.8: S-Function Builder 04.a (input ports)

For this example, the following output ports shall be added:

1. **Acceleration sensor**:
   Port name: `acc`
   Rows: `3` (X,Y,Z axis)

2. **Gyrosensor**:
   Port name: `gyro`
   Rows: `3` (roll, pitch, yaw)

3. **Magnetometer**:
   Port name: `mag`
   Rows: `3` (X,Y,Z axis)

4. **Barometer**:
   Port name: `baro`
   Rows: `1` (1-dimensional pressure value)

5. **Temperature**:
   Port name: `temp`
   Rows: `1` (1-dimensional temperature)

To add the output port to the S-Function block, click on the Add-Button as shown in fig. 11.9 (mark '1.'). A new row with a blank name will appear. Double-click on the field `Port name` and enter the first port name as shown in the enumeration above. Since the output

signal shall be a vector, the field `Dimensions` has to remain at the value '`1-D`'. Double-click on the field `Rows` and enter corresponding row number (e.g. '3' for the acceleration sensor). The field `Columns` can be left blank.

Since the IMU values are no complex numbers but completely real numbers so far, the field `Complexity` has to remain at the value `real`.

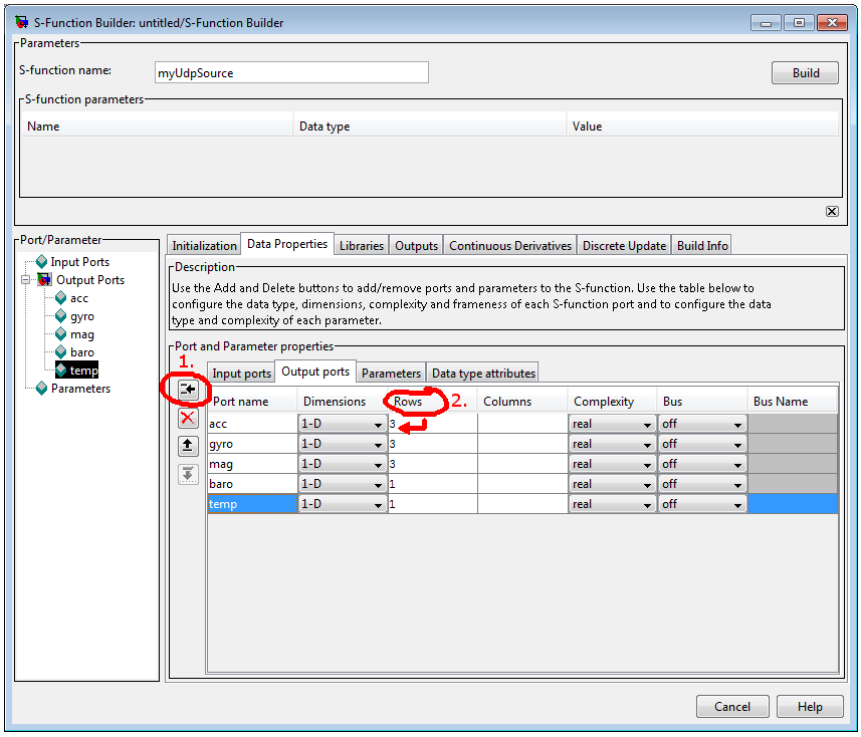


Figure 11.9: S-Function Builder 04.b (output ports)

Proceed this procedure until all output signals are defined. When all sensors are defined, the output ports list should look similar to fig. 11.9.

Simultaneously to the editing of output ports, the S-Function block in the Simulink model should change its structure of ports. When all output ports have been defined, the S-Function block should look similar to fig. 11.10. All output ports are graphically represented and properly named as given by the `Port name` in the output port list of the S-Function Builder window.

Finally, the build options can be defined. To set the build options, click on the tab `Build Info` of the main tab menu, as shown in fig. 11.11.

In this example, a network connection has to be established. For that purpose, it is necessary to call the initialization function and close function of the C library `udpLib` (see chapter 11.1). The network connection shall be established whenever the simulation is started. As soon as the Simulink simulation gets terminated, the network connection shall be closed. Therefore, the corresponding functions `Start` and `Terminate` in the S-Function

template code has to be provided by the S-Function Builder.

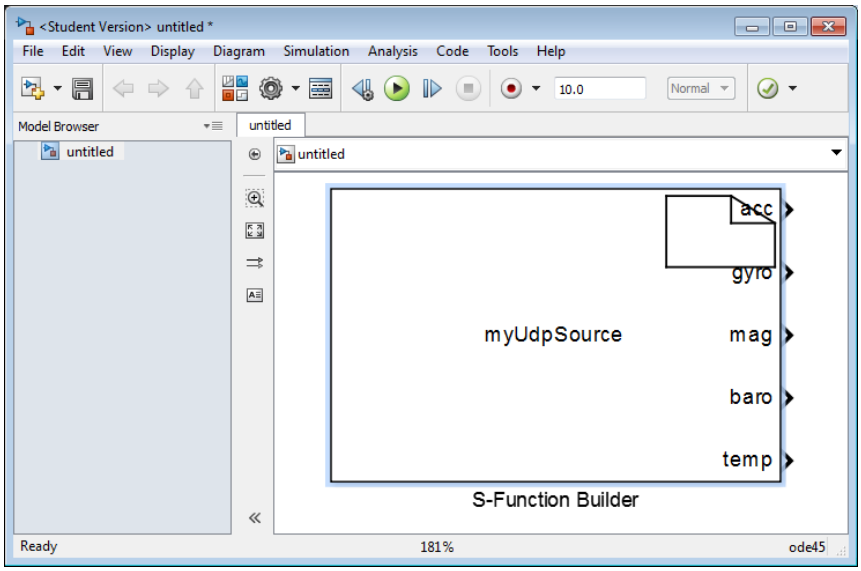


Figure 11.10: S-Function Builder 04.b (Simulink model)

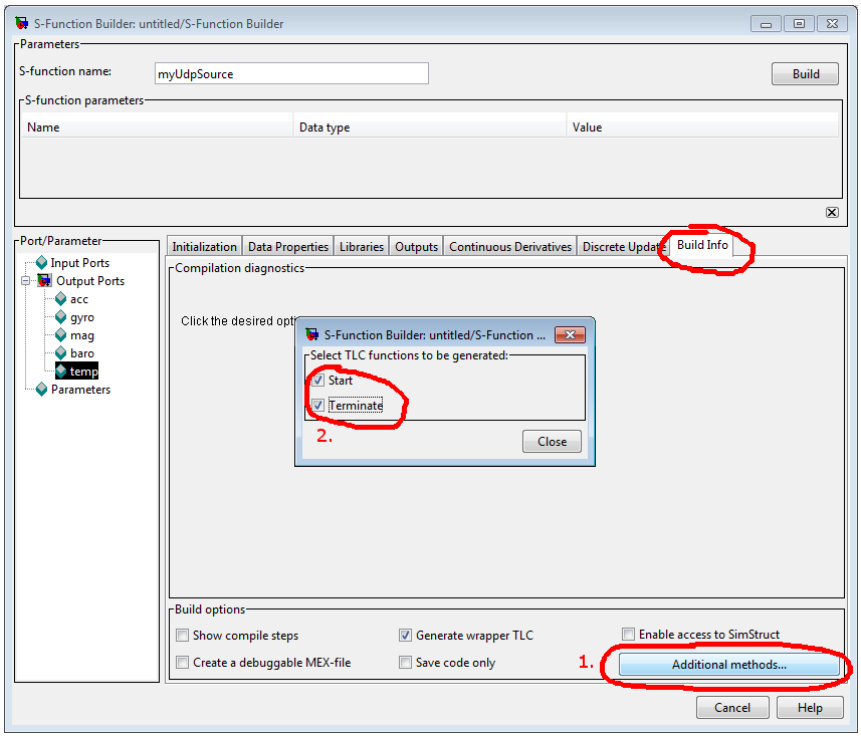


Figure 11.11: S-Function Builder 05

To activate the generation of the `Start` and `Terminate` function, click on '`Additional methods...`' at the bottom right corner of the S-Function Builder window (see fig. 11.11,

mark '1.'). A new window will pop up. Tick the check marks `Start` and `Terminate` in the menu (see fig. 11.11, mark '2.') and press the Close-button.

Optionally, there can be defined some parameters of the S-Function to parametrize the behavior of the Simulink block. To create a parameter for the S-Function, click again on tab `Data Properties` on the main tab menu (see fig. 11.12, mark '1.'). Navigate to the tab `Parameters` in the appeared menu (see fig. 11.12, mark '2.'). By default, a emtpy parameter list will appear. Press the Add-Button as depicted in fig. 11.12 (mark '3.') and a new entry will appear. Double-click on the field `Parameter name` and enter 'sampleTime'. For this example, the assumed sample time shall define expected time interval between each UDP packet received from the Raspberry Pi.

To specify a default value for the newly created parameter, click on the field `Value` of the table `S-function parameters` on the top of the S-Function Builder window (see fig. 11.12, mark '4.').

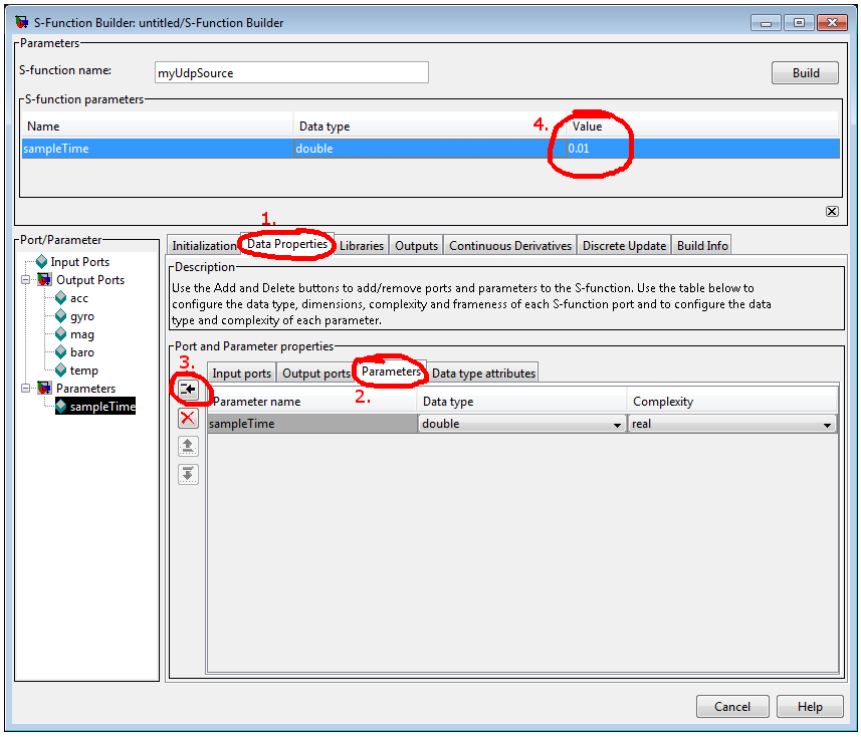


Figure 11.12: S-Function Builder 06 (optional)

Remark:
In this example, the `Data Type` for the parameter `sampleTime` has been chosen as `double`. Although it might be reasonable to choose an integer data type, for convenience and ease of use it is the best choice to use the `double` data type. MATLAB/Simulink uses the `double` data type for all signals, internally. In consequence, whenever a user will specify a number (e.g. '`100`') for a parameter MATLAB will assume to get a double value. If a different data type has been chosen in the S-Function Builder (e.g. `uint8`), the user will

have to type '`uint8(100)`'. Otherwise, a data type error will pop up and the user-defined parameter will not be accepted.

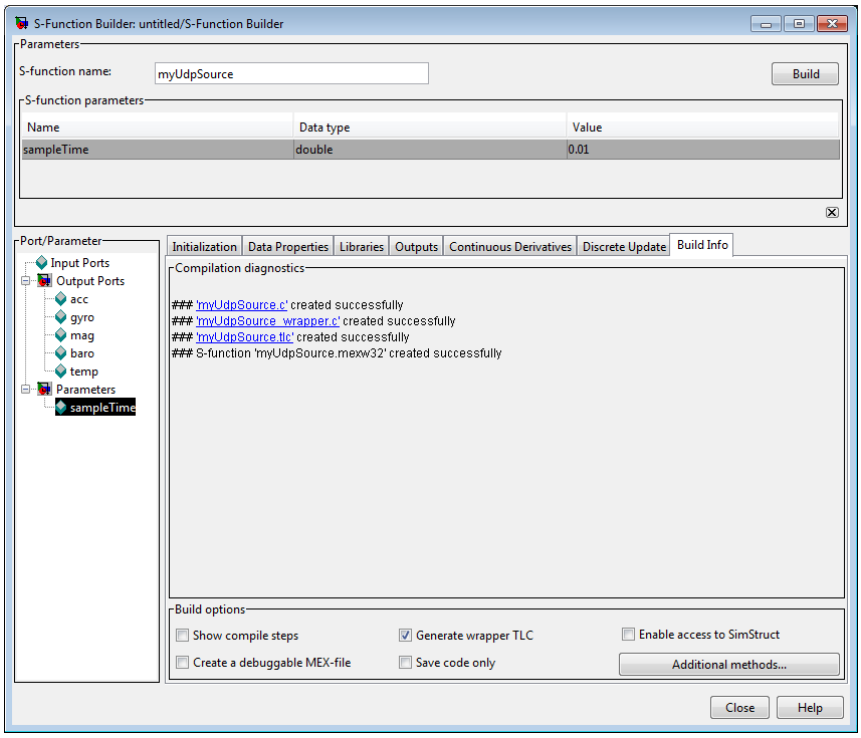


Figure 11.13: S-Function Builder 07

When all parameters and settings have been defined, the button `Build` in the top right corner of the S-Function Builder window can be pressed. The S-Function Builder will create all necessary files, such as the C-Code template files `myUdpSource.c` as well as `myUdpSource_wrapper.c`.

If the template generation was successful, the S-Function Builder window will look similar to fig. 11.13. The S-Function Builder window can be closed now, by clicking on the Close-Button in the bottom right corner.

11.2.2 Extending the C-Code templates

As a next step, the C-Code template files have to be extended to the needs of the required functionality of the S-Function block. In the case of this example, the `udpLib` library has to be included and the corresponding functions have to called in an appropriate way. A network connection shall be established whenever the Simulink simulation is started. Furthermore, the network connection shall be closed whenever the Simulink simulation has terminated. Last but not least, whenever a UDP packet is received, the data shall be extracted and transferred to the Simulink model as Simulink signals.

First, the C-Code file `myUdpSource.c` shall be edited. The complete C-Code template generated by the S-Function Builder can be found in Appendix **??**. Open the file in the preferred IDE or MATLAB.

In the top of the file, a large block comment and a long list of `define` statements can be found. According to listing **??**, after line 148 all defines have been stated. To access the network functionalities, insert here an `include` statement of the `udpImuLib` (in this example the library `udpImuLib` shall be included since all raw IMU values shall be read). Additionally to the `include` statement, two `define` statements (`REMOTE_HOST_IPADDR` and `REMOTE_HOST_PORT`) shall be inserted to define the remote host IPv4 address as well as the listen/destination port number (see line 151l. of listing 11.4). Since the udpLib establishes a symmetric UDP connection, the listen port number and the destination port number will be identical (see chapter 11.1). Lastly, an integer variable `int m_simSocketNumber` shall be declared to store the socket number of the UDP-connection.

```
143 /*<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<*/
144 #include "simstruc.h"
145 #define PARAM_DEF0(S) ssGetSFcnParam(S, 0)
146
147 #define IS_PARAM_DOUBLE(pVal) (mxIsNumeric(pVal) && !mxIsLogical(pVal) &&
        !mxIsEmpty(pVal) && !mxIsSparse(pVal) && !mxIsComplex(pVal) &&
        mxIsDouble(pVal))
148
149 #include "../path/to/udpLib/udpImuLib.h"
150
151 #define REMOTE_HOST_IPADDR  {192,168,22,161}
152 #define REMOTE_HOST_PORT    5000
153
154 static int m_simSocketNumber  = 0;
155
156 extern void myUdpSource_Outputs_wrapper(real_T *acc,
157                                         real_T *gyro,
158                                         real_T *mag,
159                                         real_T *baro,
160                                         real_T *temp,
161                                         const real_T  *sampleTime,
162                                         const int_T p_width0);
```

Listing 11.4: C-Code template file 'myUdpSource.c' extended by a include statement to link the `udpLib` library (code snippet of listing **??**, lines 143 ll.)

The changed code will look similar to listing 11.4 as shown below. In the shown code snippet, the path to the `udpImuLib` is expressed with the placeholder '`../path/to/udpLib`'. Change this placeholder to a relative path according to the folder structure used in your project.

Next, the network connection shall be established whenever the Simulink simulation gets started. Starting at line 267 of the raw C-Code template (see listing **??**) the function `mdlStart` is defined. In the function body two local variables to store the IPv4 address and the listen port number shall be defined. Furthermore, the function `g_halMatlab_initConnection_i32`

of the C library `udpLib` shall be called. The returned socket number shall be stored to `m_simSocketNumber`, as defined in listing 11.4. Optionally, a `printf` statement can be inserted for debugging purposes. The completed function body can be seen in listing 11.5 below.

```
265 #define MDL_START  /* Change to #undef to remove function */
266 #if defined(MDL_START)
267   /* Function: mdlStart
       =======================================================
268    * Abstract:
269    *    This function is called once at start of model execution. If you
270    *    have states that should be initialized once, this is the place
271    *    to do it.
272    */
273   static void mdlStart(SimStruct *S)
274   {
275     unsigned char   remoteIpAddr[4] = REMOTE_HOST_IPADDR;
276     unsigned short  listenPort      = REMOTE_HOST_PORT;
277
278     printf("Listening to port %d\n",listenPort);
279
280     m_simSocketNumber = g_halMatlab_initConnection_i32( remoteIpAddr,
281                                                         listenPort);
282   }
283 #endif /*  MDL_START */
```

Listing 11.5: C-Code template file 'myUdpSource.c' extended by the function body of the `mdlStart` function (code snippet of listing **??**, lines 265 ll.)

As a next step, the network connection shall be closed whenever the Simulink simulation gets terminated. For that purpose, the function `g_halMatlab_closeSocket_bl()` shall be called. The function parameter will be the stored socket number `m_simSocketNumber`. Depending on the boolean return state of the function `g_halMatlab_closeSocket_bl()`, a text output of the success or error shall be given to the user. The complete function body can be seen in listing 11.6 below.

```
307 /* Function: mdlTerminate
       =======================================================
308  * Abstract:
309  *    In this function, you should perform any actions that are necessary
310  *    at the termination of a simulation.  For example, if memory was
311  *    allocated in mdlStart, this is the place to free it.
312  */
313 static void mdlTerminate(SimStruct *S)
314 {
315   if ( g_halMatlab_closeSocket_bl(m_simSocketNumber) )
316   {
317     // true state refers to occurrence of errors
318     printf("Could not close the network socket\n");
319   }
320   else
321   {
```

```
322     // false state refers to absence of errors
323     printf("Network socket successfully closed\n");
324   }
325 }
```

Listing 11.6: C-Code template file 'myUdpSource.c' extended by the function body of the `mdlTerminate` function (code snippet of listing **??**, lines 307 ll.)

If the code template has been generated with custom block parameters (see chapter 11.2.1, especially fig. 11.12), the parameter `sampleTime` of this example can be used to set the default sample time of the S-Function block. Starting at line 254 of the raw C-Code template (see listing **??**) the function `mdlInitializeSampleTimes()` is defined. To adjust the default sample time to the user defined sample time, the called function `ssSetSampleTime(S, 0, SAMPLE_TIME_0)` (line 260 of listing **??**) has to be adapted. The user-defined sample time has to be extracted out of the parameter list. The complete function body of `mdlInitializeSampleTimes()` can be seen in listing 11.7 below.

```
254 /* Function: mdlInitializeSampleTimes
       =========================================
255  * Abstract:
256  *    Specifiy  the sample time.
257  */
258 static void mdlInitializeSampleTimes(SimStruct *S)
259 {
260     double* l_sampleTimePtr;
261
262     //get sample time, configured in block parameters
263     l_sampleTimePtr = mxGetPr( ssGetSFcnParam(S, 0) );
264
265     //output for debugging purposes
266     printf("Set sampling time to %lf\n",*l_sampleTimePtr);
267
268     //cast double type to time_T type
269     ssSetSampleTime(S, 0, (time_T)*l_sampleTimePtr );
270     ssSetOffsetTime(S, 0, 0.0);
271 }
```

Listing 11.7: C-Code template file 'myUdpSource.c' extended by a user-defined sample time via block parameters (code snippet of listing **??**, lines 254 ll.)

Now, the file `myUdpSource.c` is ready and fully functional to the needs of this example. The `udpLib` is successfully linked and a UDP-connection gets established and closed, accordingly to the state of the Simulink simulation. Still missing is the crucial part of receiving and translating UDP-packets to Simulink signals.

In line 302 of the raw C-Code template (see listing **??**), to translate the output signals the external function `myUdpSource_Outputs_wrapper()` gets called. This function is defined in the file `myUdpSource_wrapper.c`. Unfortunately for this example, the pre-defined interface does not fulfill all needs for the network access fnctionality.

Since a network packet shall be received inside of the outputs wrapper function, the socket number of the connection has to be known. Therefore, the function parameter list of `myUdpSource_Outputs_wrapper()` shall be extended by an integer number `int socketNum`. For this purpose, the file `myUdpSource.c` has to be altered in two lines. The declaration as well as the call of the external function `myUdpSource_Outputs_wrapper()` in line 149 and line 302, respectively, of the raw C-Code template has to be altered to the new function parameter list. The changed code-snippets of `myUdpSource.c` can be seen in listing 11.8 and listing 11.9 below.

```
149 extern void myUdpSource_Outputs_wrapper(real_T *acc,
150                                         real_T *gyro,
151                                         real_T *mag,
152                                         real_T *baro,
153                                         real_T *temp,
154                                         const real_T  *sampleTime,
155                                         const int_T p_width0,
156                                         int socketNum);
```

Listing 11.8: C-Code template file 'myUdpSource.c' with changed declaration of the external function `myUdpSource_Outputs_wrapper()` (code snippet of listing **??**, lines 149 ll.)

```
289 /* Function: mdlOutputs
    =======================================================
290  *
291 */
292 static void mdlOutputs(SimStruct *S, int_T tid)
293 {
294     real_T         *acc  = (real_T *)ssGetOutputPortRealSignal(S,0);
295     real_T         *gyro  = (real_T *)ssGetOutputPortRealSignal(S,1);
296     real_T         *mag  = (real_T *)ssGetOutputPortRealSignal(S,2);
297     real_T         *baro  = (real_T *)ssGetOutputPortRealSignal(S,3);
298     real_T         *temp  = (real_T *)ssGetOutputPortRealSignal(S,4);
299     const int_T    p_width0  = mxGetNumberOfElements(PARAM_DEF0(S));
300     const real_T   *sampleTime  = (const real_T *)mxGetData(PARAM_DEF0(S));
301
302     myUdpSource_Outputs_wrapper(acc,
303                                 gyro,
304                                 mag,
305                                 baro,
306                                 temp,
307                                 sampleTime,
308                                 p_width0,
309                                 m_simSocketNumber);
310 }
```

Listing 11.9: C-Code template file 'myUdpSource.c' with changed call of the external function `myUdpSource_Outputs_wrapper()` inside the function body of function `mdlOutputs` (code snippet of listing **??**, lines 289 ll.)

Finally, the definition of the function `myUdpSource_Outputs_wrapper()` in the C-Code file `myUdpSource_wrapper.c` has to be adapted to meet the new interface definition as

well as implement required network access functionality.

In listing 11.10, the `include` statement has been added to the file `myUdpSource_wrapper.c` to access the network functions of the `udpImuLib` library. In the shown code snippet, the path to the `udpImuLib` is expressed with the placeholder '`../path/to/udpLib`'. Change this placeholder to a relative path according to the folder structure used in your project.

```
36 /* %%%-SFUNWIZ_wrapper_includes_Changes_BEGIN --- EDIT HERE TO _END */
37 #include <math.h>
38 #include "../path/to/udpLib/udpImuLib.h"
39 /* %%%-SFUNWIZ_wrapper_includes_Changes_END --- EDIT HERE TO _BEGIN */
```

Listing 11.10: C-Code template file 'myUdpSource_wrapper.c' with added include statement for `udpImuLib` library (code snippet of listing **??**, lines 36 ll.)

In listing 11.11, the outputs wrapper function `myUdpSource_Outputs_wrapper()` is extended by the call of the network access function `g_halMatlab_recvImuState_bl()` of the `udpImuLib` library. This function receives a UDP packet and stores the payload in a structured variable. With this variable, all sent data can be easily translated to the Simulink signals.

Since the Simlink port `acc`, `gyro` and `mag` are output vectors with three rows (as defined in chapter 11.2.1) the variables can be access as C-Arrays. This can be seen in listing 11.11, line 82 ll.

```
49 /*
50  * Output functions
51  *
52  */
53 void myUdpSource_Outputs_wrapper(real_T *acc,
54                                  real_T *gyro,
55                                  real_T *mag,
56                                  real_T *baro,
57                                  real_T *temp,
58                                  const real_T  *sampleTime,
59                                  const int_T p_width0,
60                                  int socketNum)
61 {
62 /* %%%-SFUNWIZ_wrapper_Outputs_Changes_BEGIN --- EDIT HERE TO _END */
63
64     // define payload structure of UDP packets
65     halMatlab_rtImuPayload l_udpPayload_st;
66
67     // get a new packet
68     l_udpPayload_st = g_halMatlab_recvImuState_bl( socketNum );
69
70     // print remote system time for debugging purposes here
71     printf("Remote time: %ld.%ld\n",
72             l_udpPayload_st.timestamp_st.tv_sec,
73             l_udpPayload_st.timestamp_st.tv_nsec);
74
75     /* translate the UDP packet data to simulink signals
```

```
76      * Remark: Since all used data of the UDP packets are
77      *         already of the type double (see f64 postfix)
78      *         the data is already matching the double
79      *         type data representation, used by MATLAB
80      *         internally.
81      */
82     acc[0] = l_udpPayload_st.imuState_st.acc.x_f64;
83     acc[1] = l_udpPayload_st.imuState_st.acc.y_f64;
84     acc[2] = l_udpPayload_st.imuState_st.acc.z_f64;
85
86     gyro[0] = l_udpPayload_st.imuState_st.gyro.l_roll_f64;
87     gyro[1] = l_udpPayload_st.imuState_st.gyro.l_pitch_f64;
88     gyro[2] = l_udpPayload_st.imuState_st.gyro.l_yaw_f64;
89
90     mag[0] = l_udpPayload_st.imuState_st.mag.x_f64;
91     mag[1] = l_udpPayload_st.imuState_st.mag.y_f64;
92     mag[2] = l_udpPayload_st.imuState_st.mag.z_f64;
93
94     baro[0] = l_udpPayload_st.imuState_st.pressure_f64;
95
96     temp[0] = l_udpPayload_st.imuState_st.temperature_f64;
97 /* %%%-SFUNWIZ_wrapper_Outputs_Changes_END --- EDIT HERE TO _BEGIN */
98 }
```

Listing 11.11: C-Code template file 'myUdpSource_wrapper.c' with extended and implemented wrapper function `myUdpSource_Outputs_wrapper()` (code snippet of listing **??**, lines 49 ll.)

Now all the C-Code of the S-Function block has been adapted according to the requirements of this example. As a final step, the S-Function shall be compiled in MATLAB.

11.2.3 Compiling the S-Function block

In order to use the custom S-Function block, created in chapter 11.2.1 and chapter 11.2.2, so far, the C-Code has to be compiled with the MATLAB Compiler `mex`.

For that purpose, change to the MATLAB main window and change current working directory to the location of the S-Function's C-Code files. This can be done either by navigating through the `Current Folder` window or by typing '`cd ../path/to/sFuncCode`' in MATLAB's `Command Window`.

If the MATLAB compiler never has been used on the system, it might be necessary to configure the compilation environment in MATLAB first. To do so, type `mex -setup` in the `Command Window`. Plenty of text will appear, similar to listing 11.12 below. MATLAB will ask if it shall search for installed compatible compilers. Answer that question with `yes`.

```
1 >> mex -setup
2
3 Welcome to mex -setup.  This utility will help you set up
```

```
a default compiler.  For a list of supported compilers, see
http://www.mathworks.com/support/compilers/R2013a/win32.html

Please choose your compiler for building MEX-files:

Would you like mex to locate installed compilers [y]/n? y
```

Listing 11.12: Configuration of MATLAB's mex compiler (part 1)

If the used host system is a Windows Operating System and a Microsoft Visual Studio Compiler has been found (as shown in the listing 11.13 below), it is highly recommended to use the Microsoft Visual Studio Compiler for performance and ease of use purposes. Furthermore, it is possible to attach the Microsoft Visual Studio Compiler to the custom S-Function Code (Debugger gets actually attached to `Matlab.exe`) and debug the C-Code during run-time.

```
Select a compiler:
[1] Lcc-win32 C 2.4.1 in C:\MATLAB\R2013A~1\sys\lcc
[2] Microsoft Visual C++ 2010 in C:\Program Files (x86)\Microsoft Visual
    Studio 10.0

[0] None

Compiler: 2

Please verify your choices:

Compiler: Microsoft Visual C++ 2010
Location: C:\Program Files (x86)\Microsoft Visual Studio 10.0

Are these correct [y]/n? y

************************************************************************

  Warning: MEX-files generated using Microsoft Visual C++ 2010 require
           that Microsoft Visual Studio 2010 run-time libraries be
           available on the computer they are run on.
           If you plan to redistribute your MEX-files to other MATLAB
           users, be sure that they have the run-time libraries.
************************************************************************



Trying to update options file: C:\Users\aUserName\AppData\Roaming\
    MathWorks\MATLAB\R2013a\mexopts.bat
From template:              C:\MATLAB\R2013A~1\bin\win32\mexopts\
    msvc100opts.bat

Done . . .

************************************************************************
  Warning: The MATLAB C and Fortran API has changed to support MATLAB
```

```
32             variables with more than 2^32-1 elements.  In the near future
33             you will be required to update your code to utilize the new
34             API. You can find more information about this at:
35             http://www.mathworks.com/help/matlab/matlab_external/upgrading-
    mex-files-to-use-64-bit-api.html
36             Building with the -largeArrayDims option enables the new API.
37 ************************************************************************
38
39 >>
```

Listing 11.13: Configuration of MATLAB's mex compiler (part 2)

After the MEX compiler has been successfully configured, the S-Function code can be compiled. For this purpose, type the command `mex` into the `Command Window`, followed by each C-Code file used by the S-Function code (also included libraries)! For the S-Function created in this example, the complete call of the MEX compiler will look like shown below.

```
1 >> mex myUdpSource.c myUdpSource_wrapper.c ../path/to/udpLib/*.c
```

11.3 3D-Representation of orientation data

Since this UDP-based network connection to MATLAB is mainly used for testing and validation purposes, a graphical 3D output of the orientation angles produced by the functional unit `orientation` (see also chapter **??**) was required.

To visualize such a 3D representation of the Quadrocopter's orientation angles in space, the authors of this document have chosen a 3-dimensional plane that can be rotated in each rotational axis in space, representing the pitch, roll and yaw angles. For that purpose, a separate User-defined Simulink was created - based on a MATLAB-Function Block. In contrast to a S-Function block, the behavior of a MATLAB-Function block is defined by MATLAB code instead of C-Code. That simplifies the implementation of a specifically needed behavior, e.g. for Rapid Prototyping, at the cost of performance.

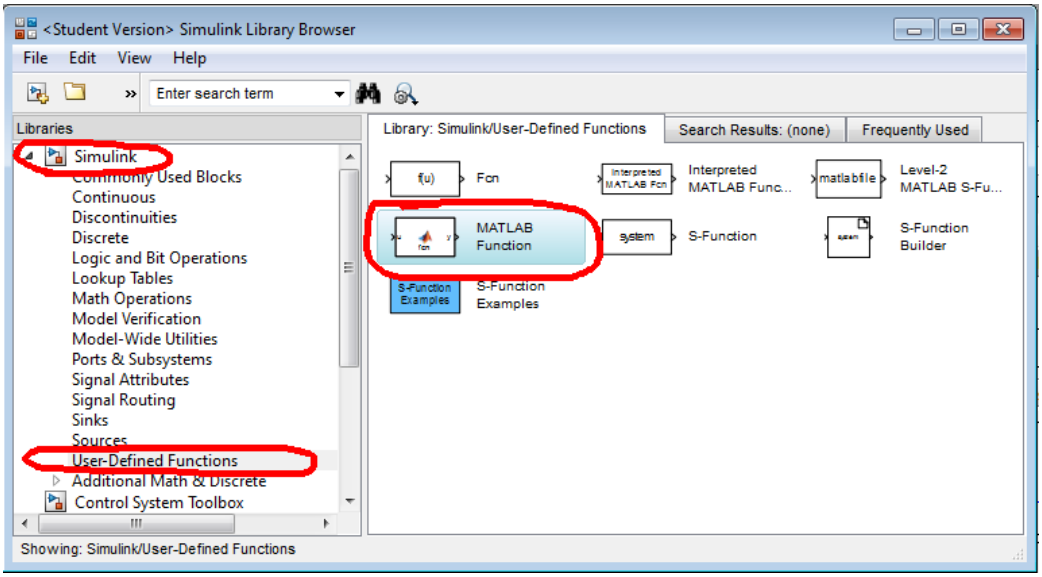


Figure 11.14: MATLAB-Function Block in Simulink Library Browser

The create a 3D visualization of the orientation angles, open a Simulink model. Open the Simulink Library Browser and click on `Simulink -> User-Defined Functions`, as depicted in fig. 11.14. Now drag the MATLAB-Function icon to the Simulink model under development.

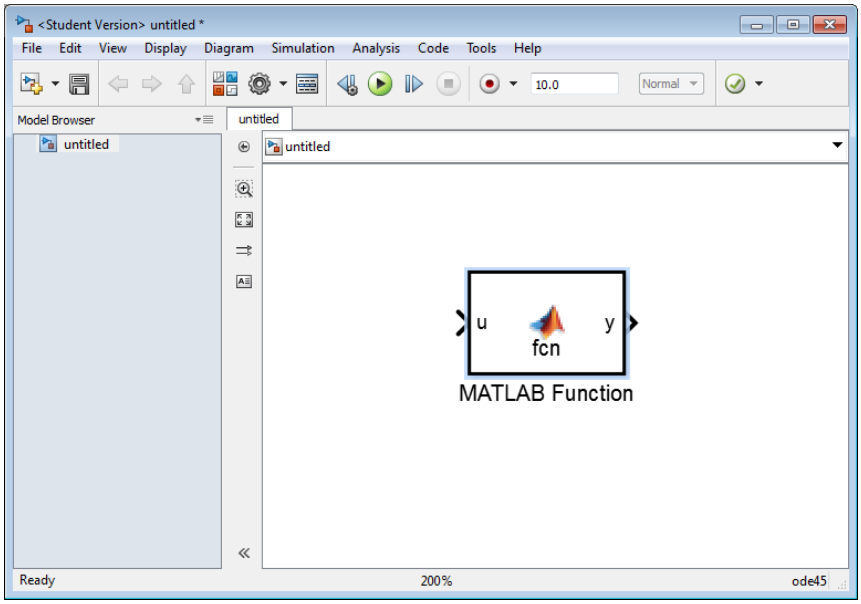


Figure 11.15: MATLAB-Function Block in a Simulink model

When the MATLAB-Function has been dragged into the model, the Simulink window should look similar to fig. 11.15. Double-click on the MATLAB Function block. A new

MATLAB code section will appear in the MATLAB Editor with an empty function body. Copy the complete code of listing **??** in the Appendix **??** and paste it into recently popped up MATLAB Editor, replacing the default code. Press the save button and check the Simulink model for any changes. The MATLAB Function block should now look similar to fig. 11.16 and is called `drawBox`, according to the function name of the copied MATLAB code of listing **??**.

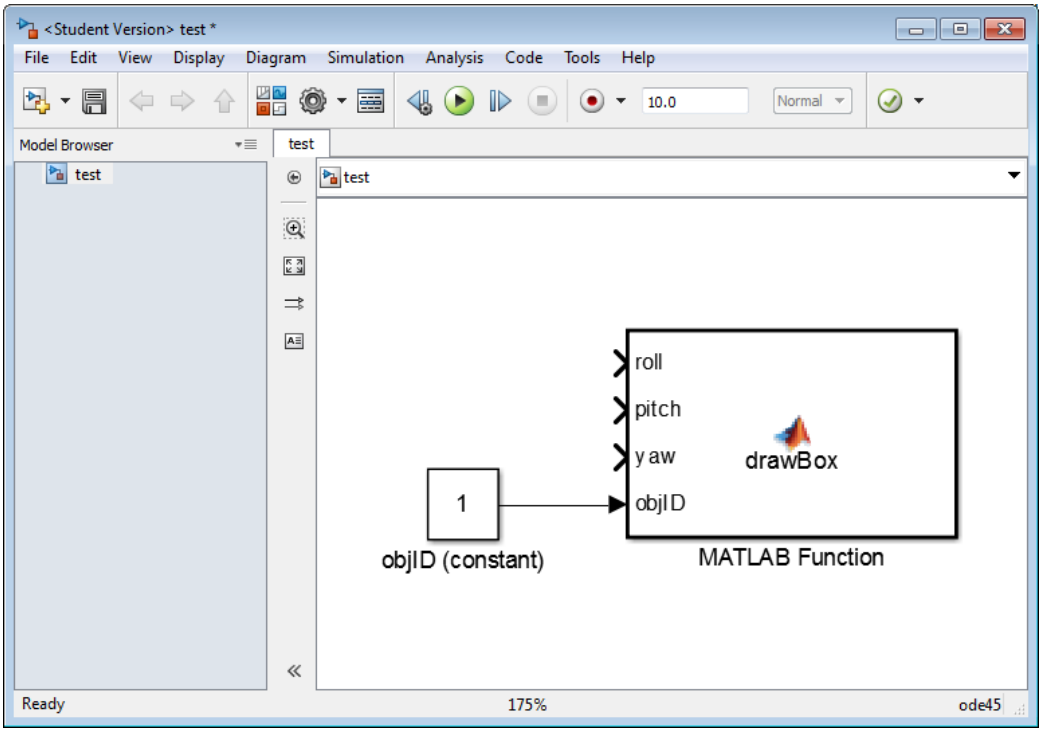


Figure 11.16: MATLAB-Function Block for 3D visualization in a Simulink model

As already depicted in fig. 11.16, the port `objID` needs to be connected to a constant value block. The constant value has to be a unique number in the range 1 to 3 (the max. range is defined by variable `objIDMax` of listing **??**, line 24). This number has to be unique among all identical `drawBox` blocks. This is needed for the internal identification of the window ID and plot ID.

Now the remaining inputs `roll`, `pitch` and `yaw` can be connected to some signal source, providing an angular value with the unit `degree`. When the model is run, an graphical output as shown in fig. 11.17 will appear. According to the angular values of the input ports `roll`, `pitch` and `yaw`, the 3-dimensional plane will rotate as shown in fig. 11.18

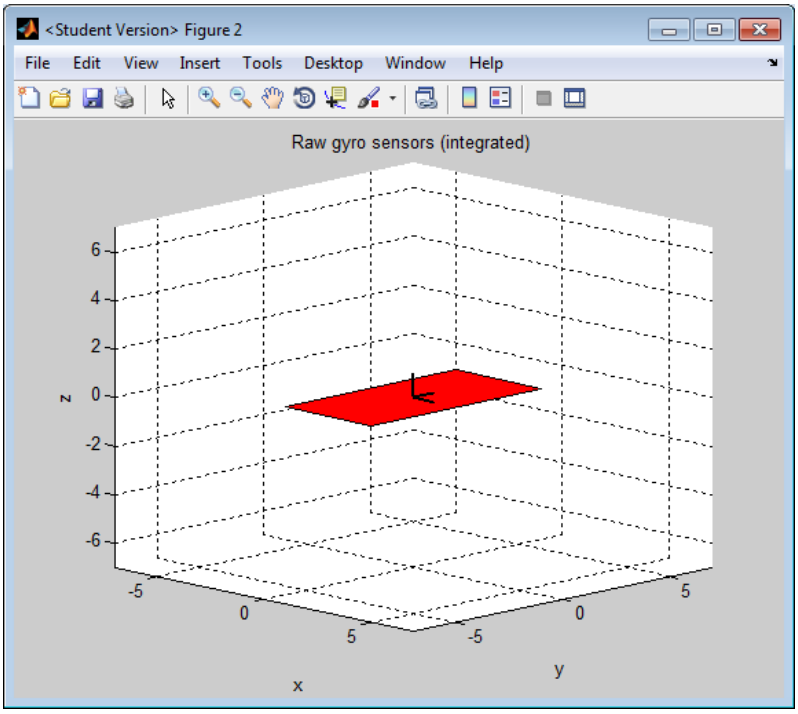


Figure 11.17: 3D graph of the MATLAB-Function Block for 3D visualization

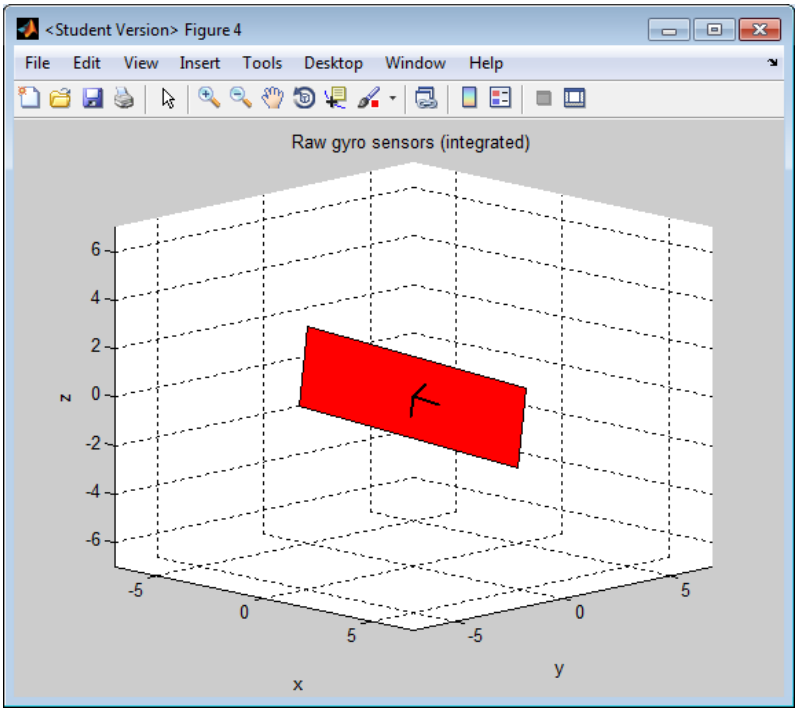


Figure 11.18: 3D graph of the MATLAB-Function Block for 3D visualization (in motion)

12 Analysing Data and sensor fusion

12.1 Calculation of the orientation angles

The Kalman filter and the Complementary filter both need angles as input. Those angles are calculated with the acceleration sensor and with the gyroscope. The gyroscope sensor is used as main input and is corrected by the acceleration/magnetic sensor. The reason why both are used is already described in the Sensor fusion chapter.
To achieve an angle from the gyroscope, the values from the sensor needs to be integrated. The following formulas show the integration part of the function of the gyroscope which runs on the Raspberry Pi. It is located in the folder 'sig/Orientientation/' in the file Orientation.c in the function 'm_sigOri_calcGyroAnglePerStep_st()'. This function calculates the angle which since the last call is made.

$$roll_{angle} = roll_{rate} * deltaT \tag{12.1}$$
$$pitch_{angle} = pitch_{rate} * deltaT \tag{12.2}$$
$$yaw_{angle} = yaw_{rate} * deltaT \tag{12.3}$$

To make this code independent from any time step, it calculates locally the time difference 'deltaT' between the last call. To make this possible the 'gettimeofday' function is used. Because of that any jitter will not make a problem. Also the gyroscope sensor gets automatically offset corrected when the code is started.

To get an angle from the acceleration/magnetic sensor the following calculation is needed. It is located in the function 'm_sigOri_calcAccMagAngle_st()' in the same file like mentioned before.
First the roll angle is calculated:

$$roll_{angle\_rad} = atan2\left(\frac{acc_{y\_axis}}{acc_{z\_axis}}\right) \tag{12.4}$$
$$roll_{angle\_deg} = -roll_{angle\_rad} * 180/Pi \tag{12.5}$$

Then the pitch is calculated:

$$pitch_{angle\_rad} = atan\left(-\frac{acc_{x\_axis}}{acc_{y\_axis} * sin(roll_{angle\_rad}) + acc_{z\_axis} * cos(roll_{angle\_rad})}\right) \tag{12.6}$$

$$pitch_{angle\_deg} = -pitch_{angle\_rad} * 180/Pi \tag{12.7}$$

Last step is the calculation of the yaw angle. The yaw angle can not be calculated by the acceleration sensor, only the magnetic sensor can be used for this. But the magnetic sensor can not directly be used. Additional a magnetic tilt compensation needs to be done. The following formulas show the calculation steps.

$$divider = mag_{x\_axis} * cos(pitch_{angle\_rad}) + \tag{12.8}$$

$$mag_{y\_axis} * sin(pitch_{angle\_rad}) * sin(roll_{angle\_rad}) + \tag{12.9}$$

$$mag_{z\_axis} * sin(pitch_{angle\_rad}) * cos(roll_{angle\_rad}) \tag{12.10}$$

$$yaw_{angle\_rad} = atan2\left(\frac{mag_{z\_axis} * sin(roll_{angle\_rad}) - mag_{y\_axis} * cos(roll_{angle\_rad})}{divider}\right) \tag{12.11}$$

$$yaw_{angle\_deg} = yaw_{angle\_rad} * 180/Pi \tag{12.12}$$

With this, the implementation of the fusion filter were done and proved. First the results seem to be correct like can be seen in figure 12.1.

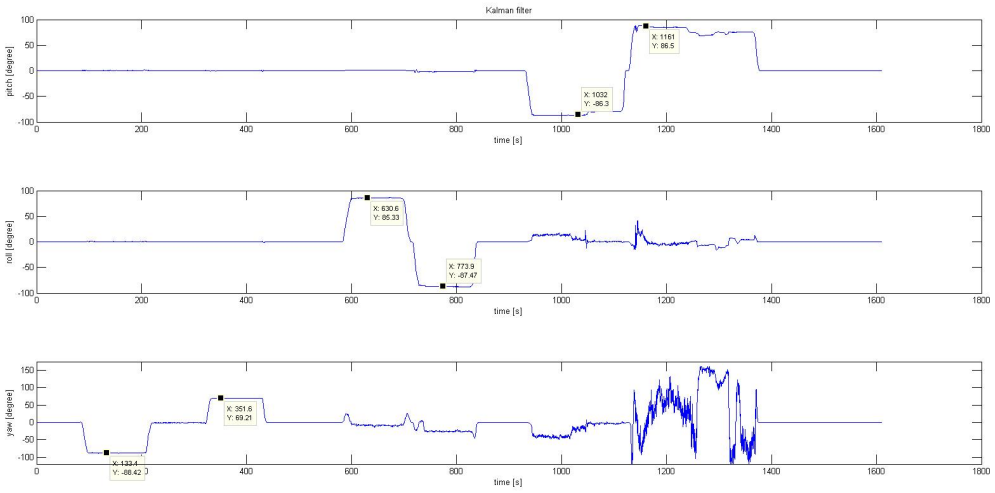


Figure 12.1: First result Kalman filter

12.1.1 Magnetic sensor test

As can be seen there is a problem with the yaw angle when a pitch angle is applied. This is no solution which can be used in the Quadrocopter. So additional tests are made. After some time the problem was detected. Because just the yaw angle makes some problem, the main observation layed on the magnetic sensor. Figure 12.2 shows the measured magnetic field.

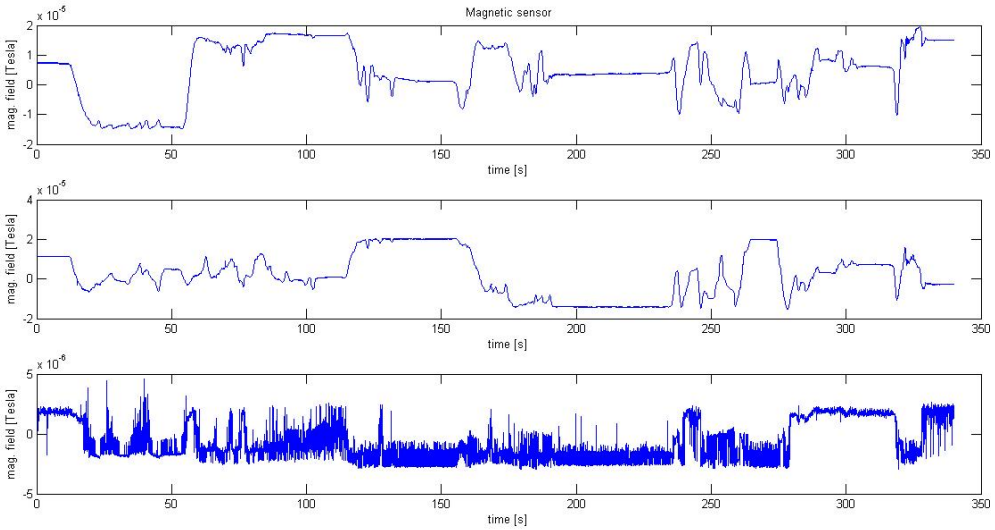


Figure 12.2: Weak magnetic field strength

The problem can directly be seen. The measured field strength should be nearly the same on all axis. The field strength on the x-axis and y-axis is the same. The z-axis delivers only just one tenth of the normal value. Because of that the noise on the sensor is in the same height like the signal and therefore acquiring not possible. First the position of the mounted IMU is disputed. After changing the following sensor values of the magnetic sensor can be achieved.

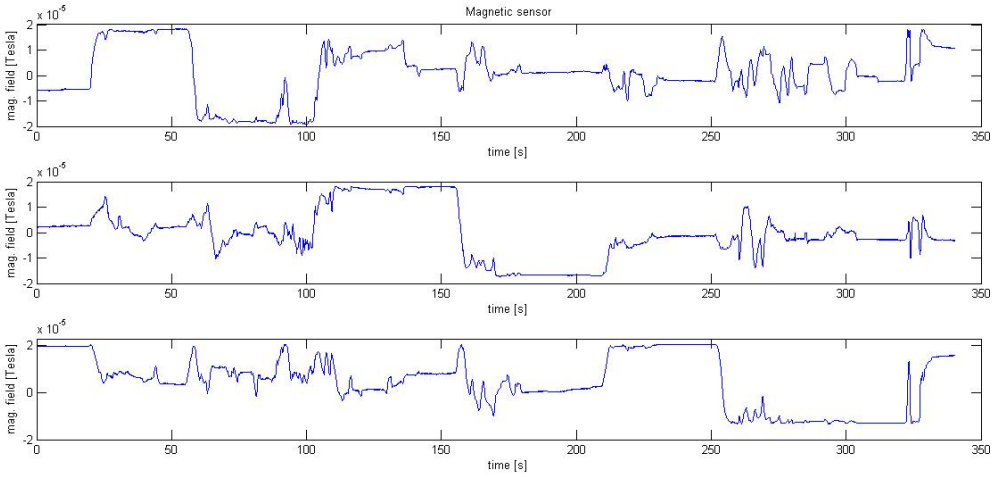


Figure 12.3: Strong magnetic field strength

By comparing the measurement of the initial positioning of the IMU with the new one, the strength on the z-axis is significantly higher. Also the strength on the three axis are nearly the same. So for further usage the new position is used.

12.1.2 Improvement magnetic sensor

The figures 12.4 and 12.5 shows the comparison of the mounting.

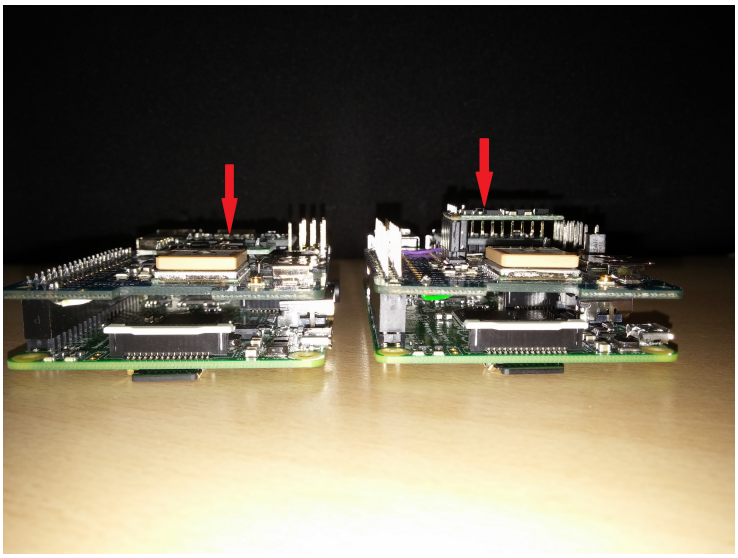


Figure 12.4: New positioning of IMU 1

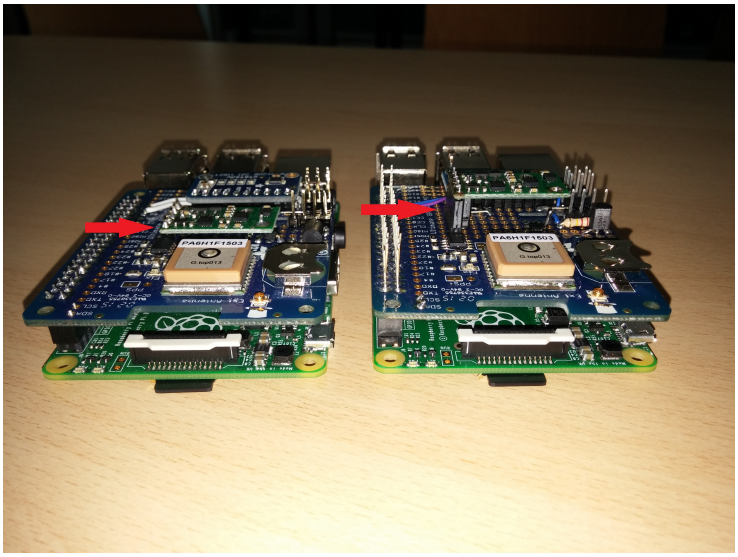


Figure 12.5: New positioning of IMU 2

The left PCB shows the old position and the right the new position. As can be seen just the height of the mounted IMU needs to be changed. Due to the tests a distance of approximately 1cm seems to be enough. So the wiring can be kept as it is.
Additionally to reduce the really high influences of near metallic parts, a hard magnetic offset compensation needs to be done. Also a scaling is done. The following calculations which run in the function 'm_sigOri_calcAccMagAngle_st()' are done within every step.

$$mag_{x\_axis} = (mag_{x\_axis} - logged_{min\_x})/(logged_{max\_x} - logged_{min\_x}) * 2 - 1 \tag{12.13}$$

$$mag_{y\_axis} = (mag_{y\_axis} - logged_{min\_y})/(logged_{min\_y} - logged_{min\_y}) * 2 - 1 \tag{12.14}$$

$$mag_{z\_axis} = (mag_{z\_axis} - logged_{min\_z})/(logged_{min\_z} - logged_{min\_z}) * 2 - 1 \tag{12.15}$$

To achieve the full range, maximum and minimum value of all three magnetic sensors the scope of Matlab can be used. First the data transmission between Matlab and the Raspberry Pi needs to be started. Then the scopes of the magnetic sensor should be opened. Now the Raspberry Pi should be rotated by parallel trying to find the maximum and minimum with the opened scopes. The m-file which is mentioned before creates directly the header-file for the code. The improved results due to the changes which are made here can be seen in the second output of the Kalman filter and complementary filter in figure 12.11 and 12.18.

12.2 Sensor fusion for Inertial Measurement Unit

To use all positive features of the sensors and reduce the negative drawbacks a sensor fusion is the needed solution. There are many possibilities for fusion algorithms. In this project a Complementary-Filter and Kalman-Filter is implemented.

For enabling a autonomous flight, the Raspberry Pi has to know the orientation. Figure 12.6 shows the axes and the naming of the rotation around the axes. These rotations are later used for the calculation for the roll, pitch and yaw angles.

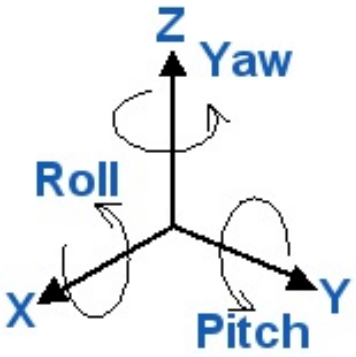


Figure 12.6: Roll, Pitch, Yaw [BorEng]

To get reliable and stable orientation of the Raspberry Pi and so from the Quadrocopter several sensors can be used. Either a acceleration sensor, a magnetic sensor or a gyroscope can be used. But each of them have some drawbacks.
The acceleration sensor is very fast and delivers reliable pitch and roll angles, but the yaw angle itself can't be calculated. Another problem is, that due to vibrations a smooth angle is not possible to calculate.

The magnetometer can be used to calculate the heading, this means the yaw angle but not the roll and pitch angle. Another problem when the sensor has a roll and pitch angle, the heading can not be easily calculated. In this case a tilt compensation has to be done. In the figure 12.7 the logging of the pitch and roll angle calculated from the acceleration sensor and the yaw angle calculated with the magnetometer can be seen. When those sensors are not combined errors occur during the calculation. In the time from 250 seconds to 450 seconds a yaw angle is applied which leads just to a yaw angle change. In the time area from 500 seconds to 600 seconds a roll angle is applied which also leads to a yaw change which is an error. In the time area from 700 seconds to 800 seconds a pitch angle is applied which also leads to a yaw change which is an error.

When designing a system using multiple MEMS sensors, it is important to understand the advantages and disadvantages of accelerometers, gyroscopes, magnetometers, and pressure sensors.

Sensor fusion solves key motion sensing performance issues of 6-axis modules consisting of a 3-axis accelerometer and a 3-axis gyroscope or a 3-axis accelerometer and a 3-axis magnetic sensor. 1) A 6-axis inertial module with an accelerometer and a gyroscope loses its absolute orientation as the gyro drifts over time, requiring calibration to restore accurate heading reference. 2) A 6-axis module with accelerometer and magnetometer is prone to data corruption in the presence of ferrous materials in the environment. 3) A 9-axis module with an accelerometer, a gyroscope and a magnetometer eliminates the drift that occurs with stand-alone sensor solutions. But these can be subject to magnetic interference. Algorithms to fuse the sensor data are required to compensate for the magnetic interference.

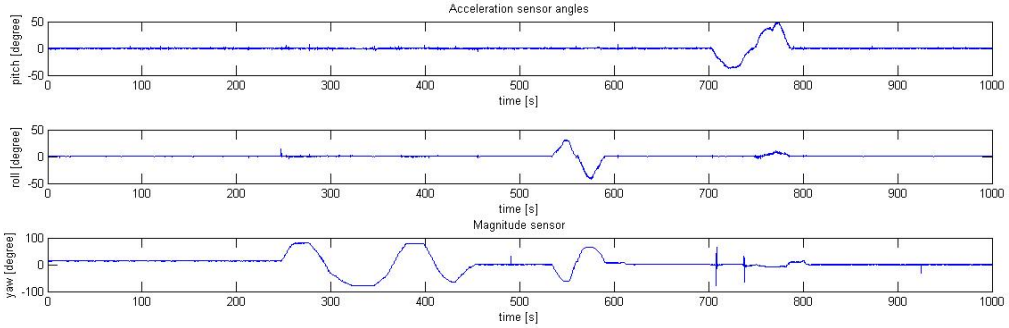


Figure 12.7: Magnetic / Acceleration angles

The last sensor is the gyroscope. The angle can be easily calculated by integrating the rates of the gyroscope. The sensor is not as fast as the acceleration sensor. So vibrations make no problems for the calculation. But due to the problem of the offset of the gyroscope, the angles will be drifting because of the integration of the rotation rates. This can be seen in figure 12.8

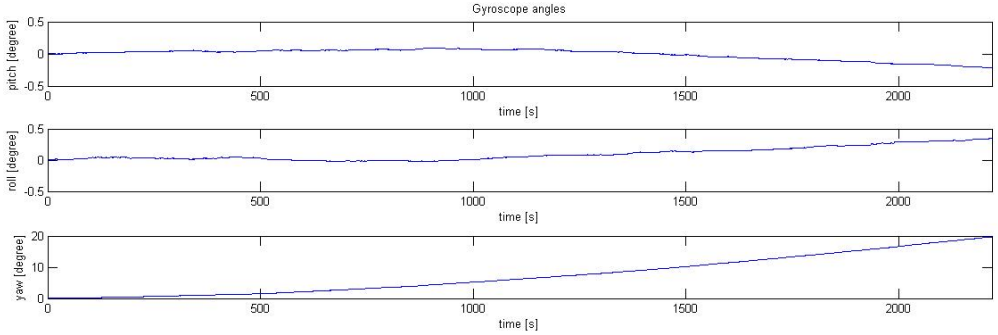


Figure 12.8: Gyroscope angles

12.2.1 Complementary-Filter

The first approach for a sensor fusion is the complementary filter. This filter uses a high-pass filter for the gyroscopes and a lowpass filter for the acceleration sensor. The highpass filter is used after the integration of the rotation rates. This can be seen in figure 12.9.

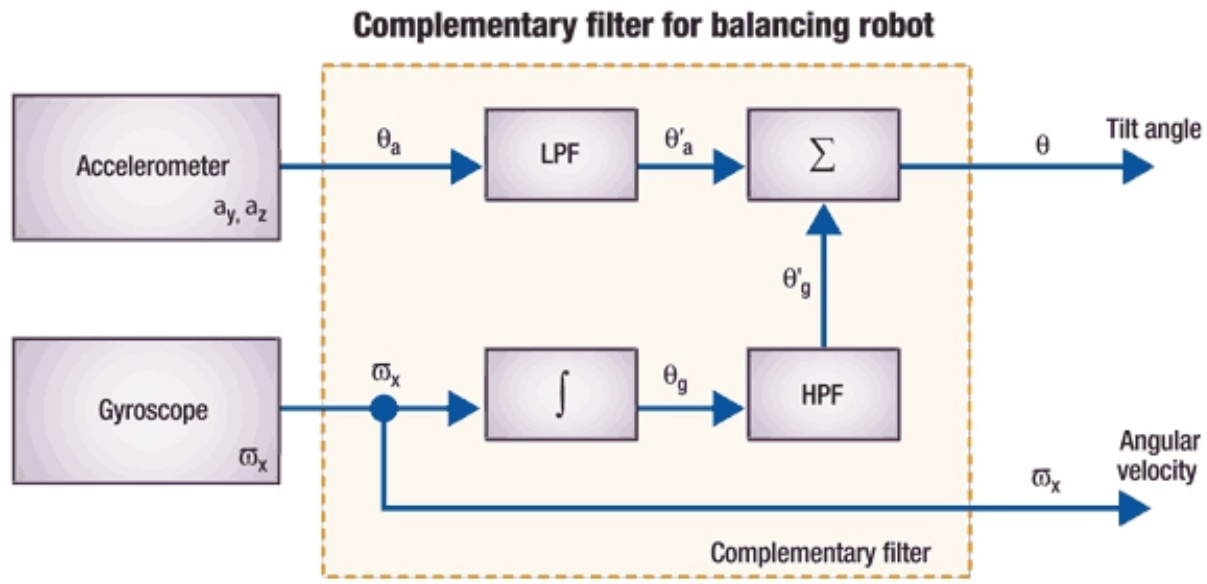


Figure 12.9: Complementary-Filter[STM]

The implementation effort is extremely lower than with the Kalman Filter. Because here no matrices and matrix operations are needed. Also are there less calculations and so fits this algorithm better to microprocessors. The only thing what needs to be checked is the lowpass/highpass-filter coefficient. Next the needed calculations are mentioned to show the difference of the complementary and Kalman filter.

- Calculation of the accelerometer and magnetometer angles
- Calculation of the gyroscope angles
- Complementary-Filter usage with defined filter time of the lowpass and highpass-filter

As written in the chapter about the sensor fusion for the complementary filter a filter time needs to be chosen. This constant is used for the lowpass filter for the acceleration/magnetic sensor and for the highpass filter of the gyroscope.

The complementary filter uses the following calculation:

$$angle = alpha * (angle + integrated\_Gyro) + (1 - alpha) * Acc\_Mag\_angle \tag{12.16}$$

As an initial guess for the first try the complementary filter uses the filter constant of 0.995. The sampling period of the sensors is 800 Hz. Usage of the filter constant of 0.995 and the sampling period of 800 Hz leads to a cut off frequency of 4 Hz.

$$alpha = \frac{time\_constant}{time\_constant + sample\_period} \tag{12.17}$$

$$alpha = 0.995 \tag{12.18}$$

$$sample\_period = \frac{1}{800Hz} = 0.00125sec \tag{12.19}$$

$$\tag{12.20}$$

This leads to:

$$time\_constant = 0.2488sec \approx \frac{1}{4Hz} \tag{12.21}$$

With this constant the following result was achieved. To make a test, first the yaw angle is changed in the range of ±90°, after that a roll angle is changed in the range of ±90° and finally the pitch angle is changed in the range of ±90°.

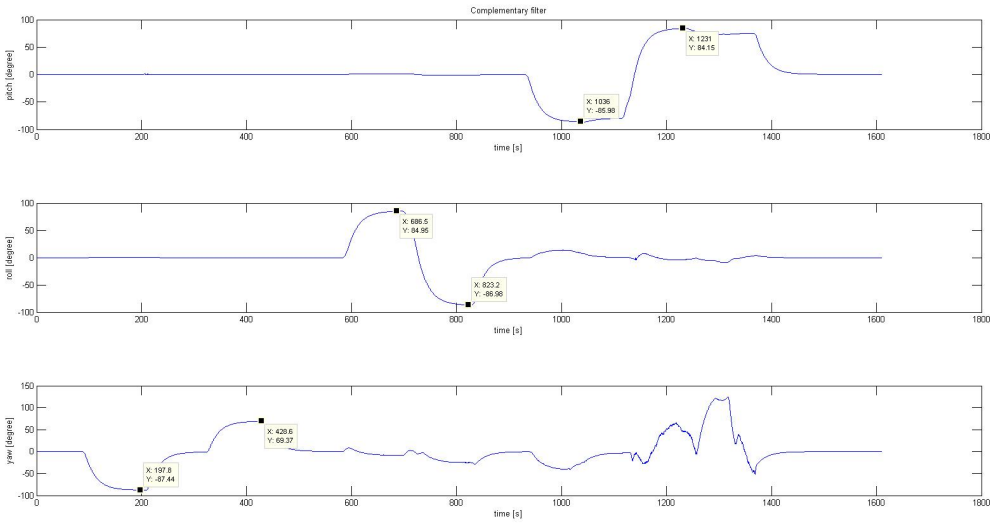


Figure 12.10: First result complementary filter

First what catches somebody's eye is that the filter is to slow. So the steady state value is achieved after consuming to much time. Also the yaw angle is not equally distributed over the whole 360 degree. In one direction the angle changes just around 70 degree. When changing roll the influences in yaw is just because the rotation was not straight in roll direction. Changing the pitch angle to much lead to a change in yaw. The next step is to improve the yaw angle, so that it will reach also the +90 degree. Also the yaw change during pitch will be observed and compared to the initial measurement. Also the time constant will be changed that a faster response can be seen.

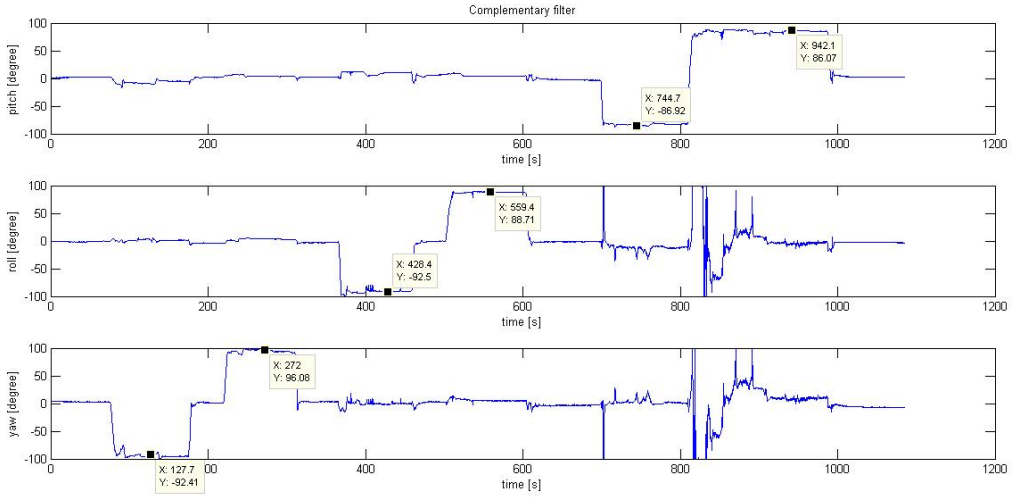


Figure 12.11: Final result complementary filter

The comparison of the first result and the final result shows the increased filter speed. The signal needs not so much time to reach the final value. Also the yaw angle reaches the 90 degree when turning 90 degree. The extreme influences on yaw directly after changing of the rotation angle results from a hand made rotation. The influences on yaw while an other angle is applied is extremely reduced.

The last figure shows how off an angle can be when just an gyroscope is used. On the left side the integrated gyroscope can be seen. On the right side, the 3D representation of the fusioned sensors are displayed.

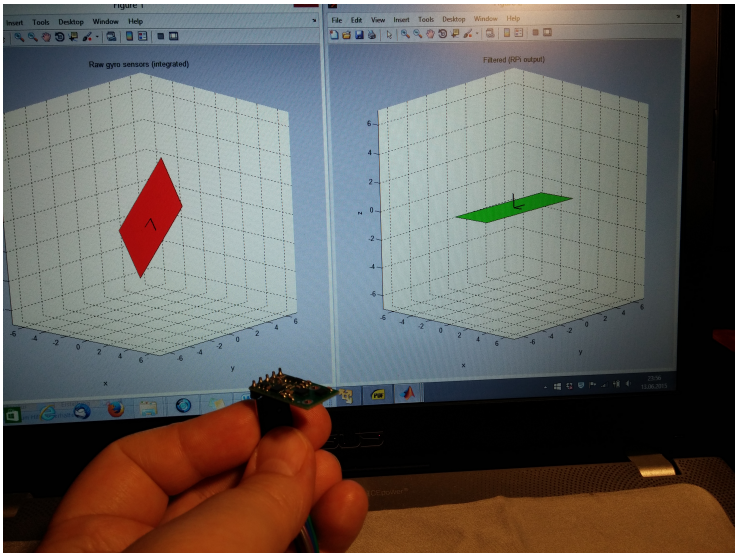


Figure 12.12: 3D representation of the integrated gyroscope raw values on the left and the fusion filtered on the right

12.2.2 Kalman-Filter

The second approach for a sensor fusion is the Kalman filter. This filter is based on the state space modeling where between the dynamics of the system and the process of the measurement is differentiated. Although the measuring is faulty and the system state is noisy, this filter guesses with the help of a set of equations the correct true state of the system. Instead of using the absolute value it uses the mean value and variance of the normal distribution. The mean value is the perfect measurement and the variance tells the uncertainty of the measurement.

The following state space description shows the calculation of the new states which in the case of this project represent the angles pitch, roll and yaw. Equation 12.22 shows the defined state vector.

$$\vec{x} = \begin{pmatrix} pitch \\ roll \\ yaw \end{pmatrix} \tag{12.22}$$

The calculation of a new state and the output can be seen in the formulas 12.23 and 12.24.

$$\vec{x_k} = \underline{A}\vec{x}_{k-1} + \underline{B}\vec{u}_{k-1} + W_{k-1} \tag{12.23}$$

$$\vec{y}_k = \underline{H}\vec{x}_k + V_k \tag{12.24}$$

Legend of the formula shown above:

- $\vec{x}_k$ state vector of actual step
- $\vec{x}_{k-1}$ state vector of previous step
- $\underline{A}$ system matrix
- $\underline{B}$ input matrix
- $\vec{u}_{k-1}$ input vector of previous step
- $\underline{W}$ process noise
- $\vec{y}_k$ output vector
- $\underline{H}_k$ output matrix
- $\underline{V}$ measurement noise
- $\underline{P}$ Output Covariance
- $\underline{Q}$ Process Noise Covariance
- $\underline{R}$ Measurement Noise Covariance

Because of the measurement and the process noise the new state is not good. The Kalman filter takes the process and measurement noise into account to improve the state estimation. To do so the Kalman filter is split into two parts, the prediction step (time update) and correction step (measurement update). In the prediction step the filter estimates the states in the next step and calculates the new covariance. Those values are calculated for the states which in the following step are expected to be reached. In the next step, the correction update, the filter checks if the pre-calculated state is reached. According to the difference the correction for the following prediction is done.

Prediction step:

Predict the next state:

$$\vec{x}_k = \underline{A}\vec{x}_{k-1} + \underline{B}\vec{u}_{k-1} \tag{12.25}$$

Predict the covariance for the next step:

$$\underline{P}_k = \underline{A}\underline{P}_{k-1}\underline{A}^T + \underline{Q} \tag{12.26}$$

Correction step:

Computation of the Kalman gain:

$$\underline{K}_k = \underline{P}_k\underline{H}^T(\underline{H}\underline{P}_k\underline{H}^T + \underline{R})^{-1} \tag{12.27}$$

Updating state prediction with new measurement:

$$\vec{x}_k = \vec{x}_k + \underline{K}_k(\vec{z}_k - \underline{H}\vec{x}_k) \tag{12.28}$$

Updating the error covariance:

$$\underline{P}_k = (\underline{I} - \underline{K}_k\underline{H})\underline{P}_k \tag{12.29}$$

These steps have to be done more than just ones, so when the correction is finished, the update has to be called again and so on.
As can be seen the implementation effort is higher than with the Complementary Filter. Because matrices and matrix operations are needed, more calculation steps are needed and the functionality of the Kalman filter is not as intuitive like with the complementary filter. Also the uncertainty of the process itself and the measurement error needs to be known. Nevertheless the results from the Kalman filter are better than those of the complementary filter. This can be seen in the results showing chapter of this project.

Figure 12.13 shows a measurement of an acceleration sensor.

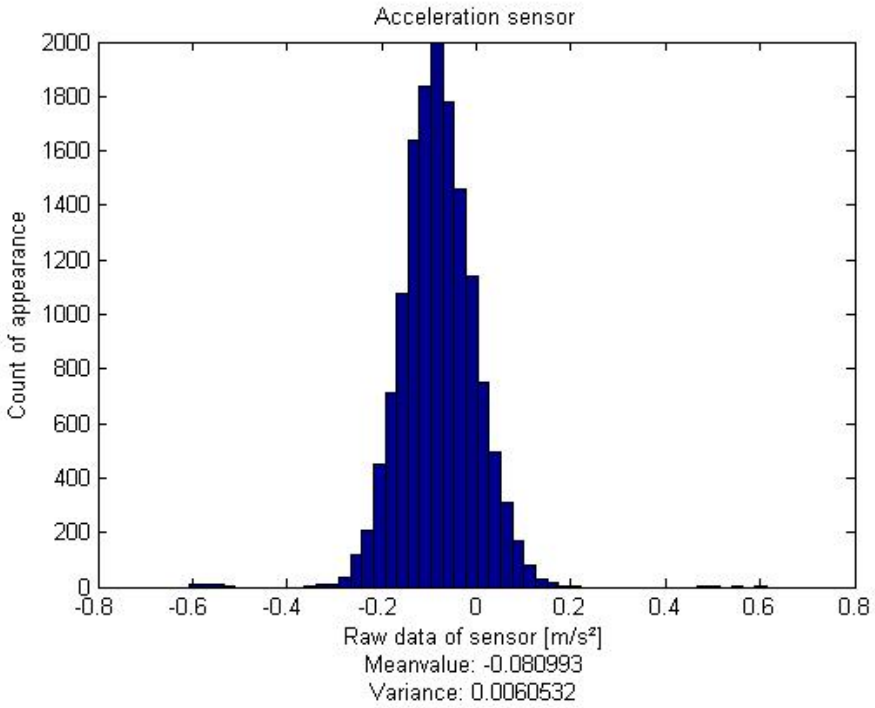


Figure 12.13: Variance and Meanvalue of an acceleration sensor

For the Kalman filter the noise of the measurement and the process has to be set up. This noise values are the variances of the used sensors and of the process. In the calculation process are those matrices named $\underline{Q}$ and $\underline{R}$. The matrix $\underline{Q}$ uses the variances of the process noise and the matrix $\underline{R}$ uses the variances of the noise of the sensors.

As an initial guess the matrices are set to:

$$\underline{Q} = \begin{pmatrix} 0.005 & 0 & 0 \\ 0 & 0.005 & 0 \\ 0 & 0 & 0.0001 \end{pmatrix} \tag{12.30}$$

$$\underline{R} = \begin{pmatrix} 0.5 & 0 & 0 \\ 0 & 0.5 & 0 \\ 0 & 0 & 0.01 \end{pmatrix} \tag{12.31}$$

With those matrices the following result was achieved. To make a test, first the yaw angle is changed in the range of ±90°, after that a roll angle is changed in the range of ±90° and finally the pitch angle is changed in the range of ±90°.

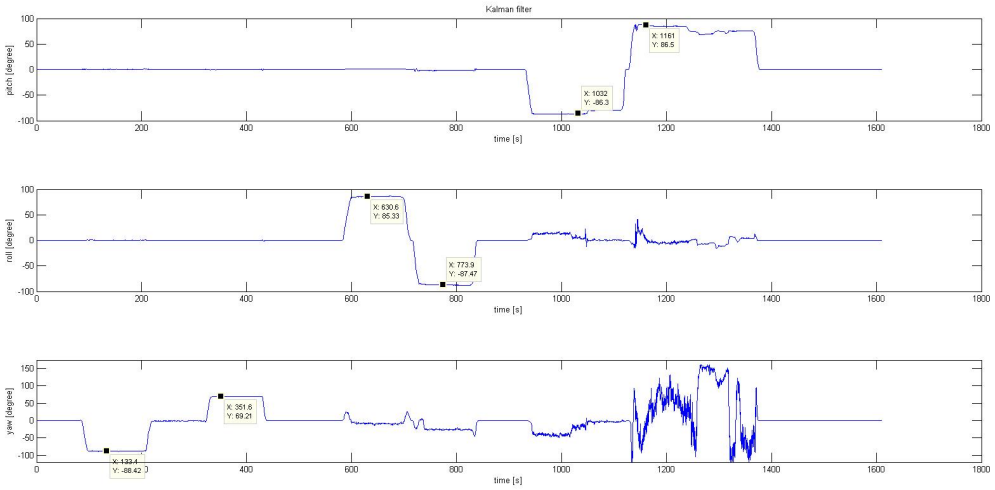


Figure 12.14: First result Kalman filter

The filter responses in a adequate time. Yaw angle change is like with the complementary filter from -90 degree to just 70 degree. The roll angle changes sufficient. But when changing the pitch angle to high a extreme yaw angle change can be seen. So the next step is to improve the yaw angle, so that it will reach also the +90 degree. Also the yaw change during pitch will be observed. After measuring the mean value and variances of all sensors they can be used to improve the Kalman filter. The figures 12.15, 12.16 and 12.17 show the logged data.

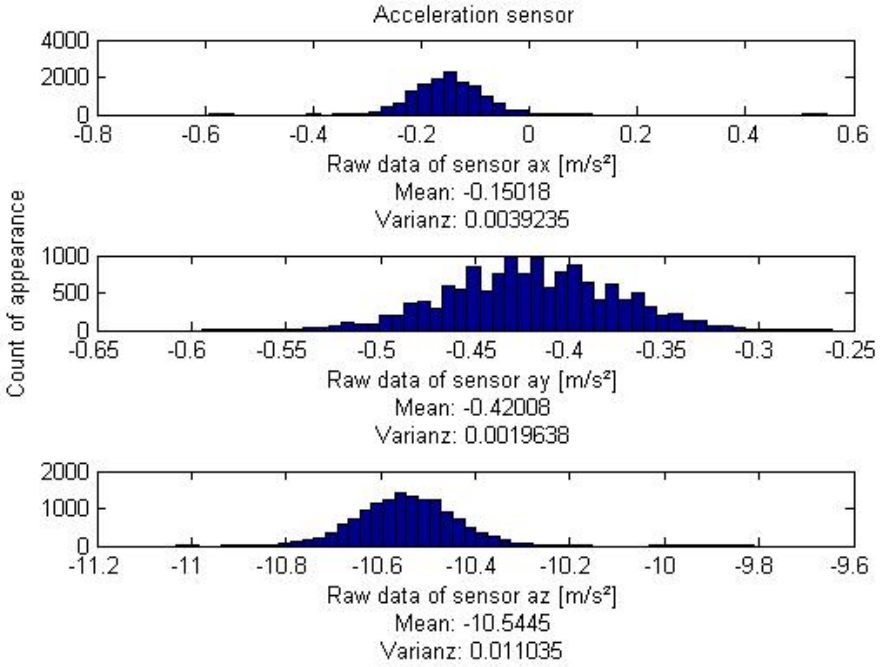


Figure 12.15: Analyzing the acceleration sensor

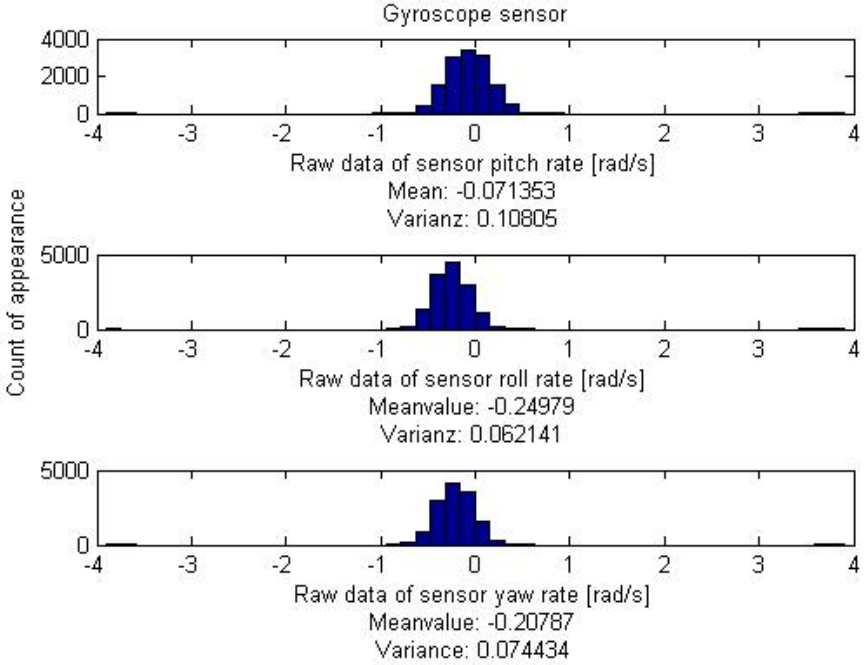


Figure 12.16: Analyzing the gyroscope sensor

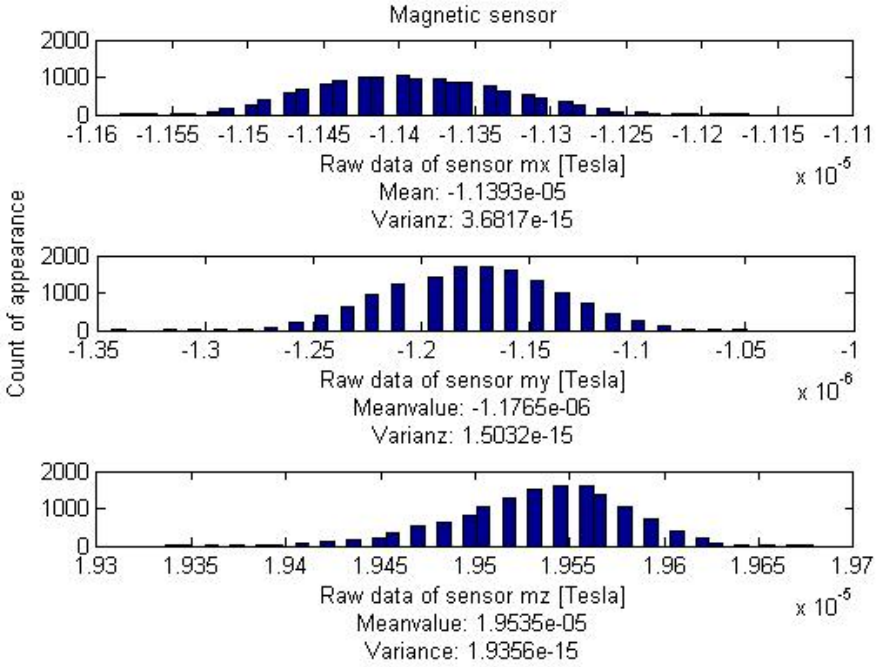


Figure 12.17: Analyzing the magnetic sensor

First the matrix $\underline{Q}$, the variances of the process noise and the matrix $\underline{R}$, the variances of the measured sensor noises are changed by using the measured values. The new values are:

$$\underline{Q} = \begin{pmatrix} 0.005 & 0 & 0 \\ 0 & 0.005 & 0 \\ 0 & 0 & 0.005 \end{pmatrix} \tag{12.32}$$

$$\underline{R} = \begin{pmatrix} 0.06 & 0 & 0 \\ 0 & 0.1 & 0 \\ 0 & 0 & 0.07 \end{pmatrix} \tag{12.33}$$

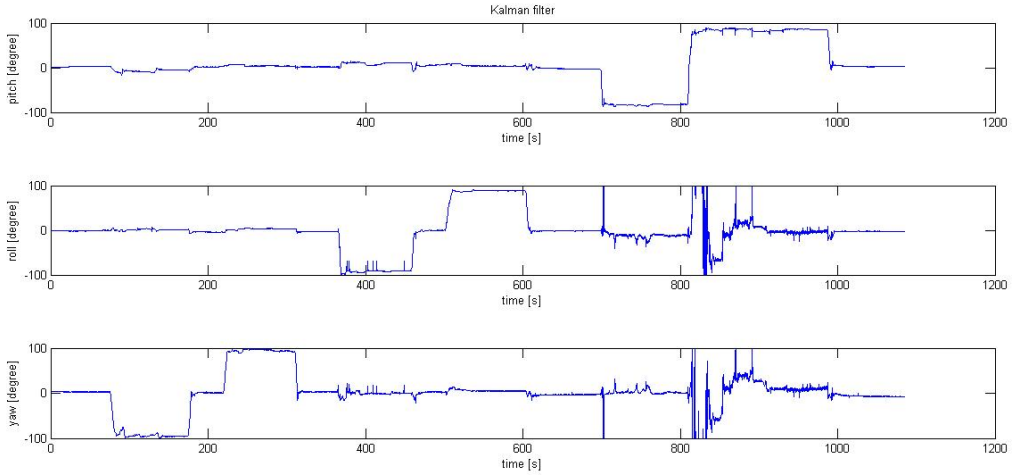


Figure 12.18: Final result Kalman filter

The wanted improving of the yaw angle in reaching the 90 degree when turning 90 degree is successfully reached. The extreme influences on yaw directly after changing of the rotation angle results from a hand made rotation. The influences on yaw while an other angle is applied is extremely reduced.

The last figure shows how off an angle can be when just an gyroscope is used. On the left side the integrated gyroscope can be seen. On the right side, the 3D representation of the fusioned sensors are displayed.

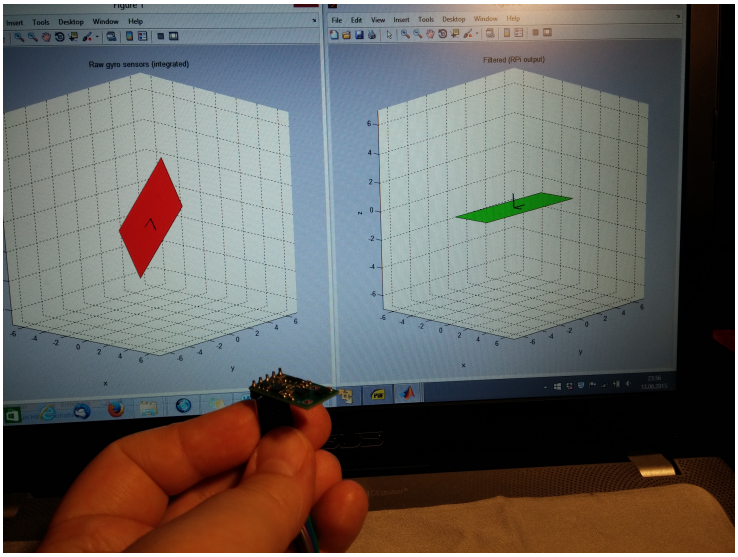


Figure 12.19: 3D representation of the integrated gyroscope raw values on the left and the fusion filtered on the right

12.3 Matrix library

Like already mentioned in the Kalman chapter various matrix operations are needed. So the simple assignment from a matrix to another, the addition and subtraction of a matrix from another. Additionally the multiplication of two matrices, the initialization of a matrix and building of the identity matrix. Finally the transpose of a matrix and the inversion of a matrix.
Assignment:

$$\underline{A} = \underline{B} \tag{12.34}$$
$$A(0,0) = B(0,0) \tag{12.35}$$
$$... \tag{12.36}$$
$$A(n,m) = B(n,m) \tag{12.37}$$

Addition/Subtraction:

$$\underline{A} = \underline{B} \pm \underline{C} \tag{12.38}$$
$$A(0,0) = B(0,0) \pm C(0,0) \tag{12.39}$$
$$... \tag{12.40}$$
$$A(n,m) = B(n,m) \pm C(n,m) \tag{12.41}$$

Multiplication:

$$\underline{A} = \underline{B} \cdot \underline{C} \tag{12.42}$$
$$A(0,0) = B(0,0) \cdot C(0,0) + ... + B(0,m) \cdot C(n,0) \tag{12.43}$$
$$... \tag{12.44}$$
$$A(n,m) = B(n,0) \cdot C(0,m) + ... + B(n,m) \cdot C(n,m) \tag{12.45}$$

Initialization:

$$A(0,0) = xx.xx \tag{12.46}$$
$$A(0,1) = xx.xx \tag{12.47}$$
$$... \tag{12.48}$$
$$A(1,0) = xx.xx \tag{12.49}$$
$$... \tag{12.50}$$
$$A(n,m) = xx.xx \tag{12.51}$$

Identity matrix:

$$A(0,0) = 1 \tag{12.52}$$
$$A(0,1) = 0 \tag{12.53}$$
$$... \tag{12.54}$$
$$A(1,0) = 0 \tag{12.55}$$
$$A(1,1) = 1 \tag{12.56}$$
$$... \tag{12.57}$$
$$A(n,n) = 1 \tag{12.58}$$

Transpose a matrix:

$$A(0,0) = A(0,0) \tag{12.59}$$
$$A(0,1) = A(1,0) \tag{12.60}$$
$$... \tag{12.61}$$
$$A(3,0) = A(0,3) \tag{12.62}$$
$$A(4,0) = A(0,4) \tag{12.63}$$
$$... \tag{12.64}$$
$$A(n,n) = A(n,n) \tag{12.65}$$

Invert a matrix:
To ensure that the inversion not only works with small matrices, the implementation uses a different way. So first the Cholesky method is used to build a lower triangular matrix. Next step is solving a linear system to get the inverse of the matrix. With this method all sizes of matrices can be inverted. It just has to be positive definite.

12.4 Sensor fusion controlling

To obtain the statistical data of all sensors of the inertial measurement unit a Matlab model is used. This model also helps to check the results of the two different fusion filters, the Kalman filter and the complementary filter. The data is send via UDP-packets from the Raspberry Pi to the Matlab model on the Host computer. Figure 12.20 shows the used Matlab model. Additional a 3D representation of the rotations is visualized and can be directly compared with the integrated offset corrected gyroscope data.

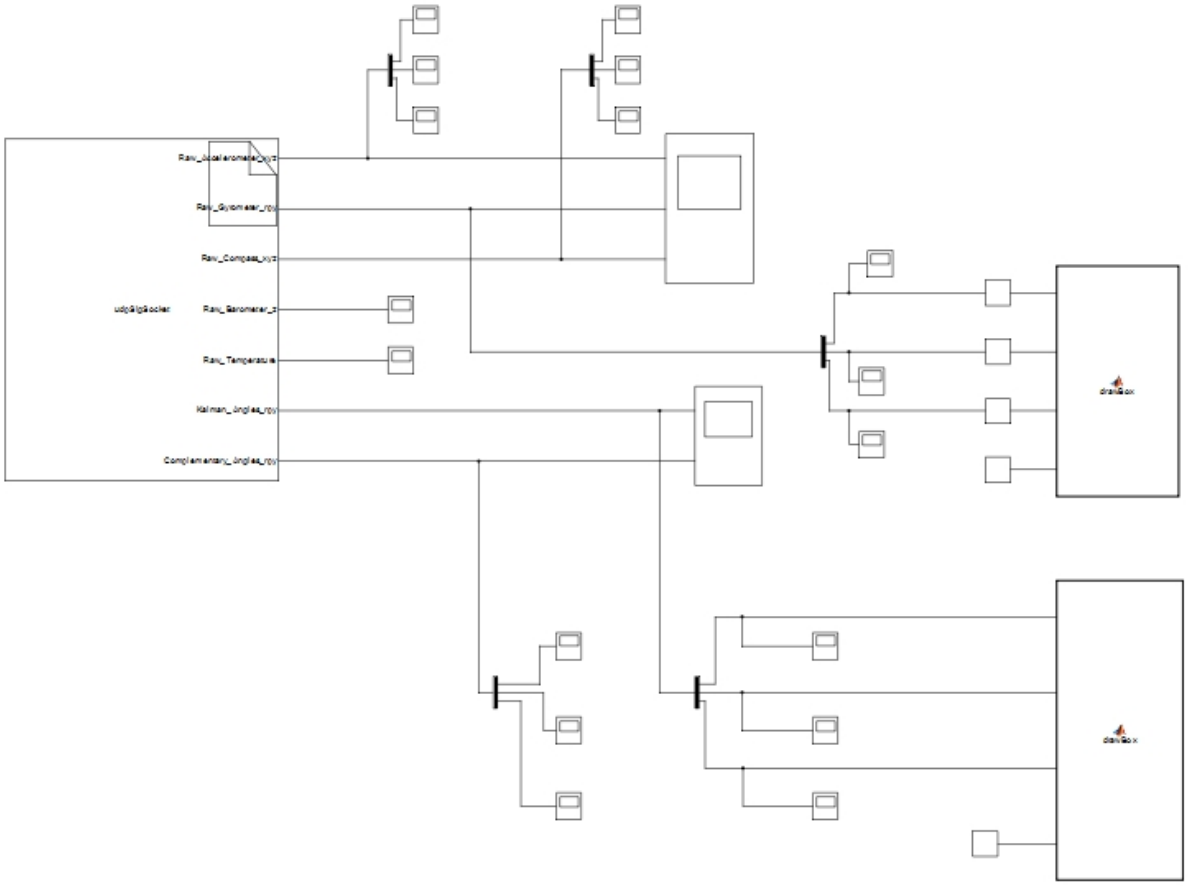


Figure 12.20: Matlab model

With the help of the scopes within this model, the logged data is stored in the Matlab workspace and then it is analyzed with a Matlab file. With the help of this file, a histogram from every sensor is generated. Also the mean value and the variance from those sensor is calculated. Finally a plot from the complementary filter and Kalman filter is generated. Also the needed minimum and maximum values of the magnetic sensor is stored. This needs to be done to reduce the influences of hard magnetic parts to the magnetic sensor. Those stored minimum and maximum values are then written in a header file (OrientationDefines.h) which needs to be copied to the folder of the Orientation.c and Orientation.h files. To achieve the best performance, the description of chapter **??** needs to be followed.

```
1  headerFileName = 'OrientationDefines.h';
2
3  %Acceleration sensor
4  figure(1)
5  %plot histogram of ax and calculate meanvalue and variance
6  subplot(3,1,1)
7  hist(ax.signals.values(:),50);
8      M=mean(ax.signals.values(:));
9      V=var(ax.signals.values(:));
10 title('Acceleration sensor')
11 disp(['Meanvalue: ' num2str(M)])
12 xlabel({'Raw data of sensor ax' ['Mean: ' num2str(M)] ['Varianz: ' num2str
       (V)]}) % x-axis label
13
14 %plot histogram of ay and calculate meanvalue and variance
15 subplot(3,1,2)
16 hist(ay.signals.values(:),50);
17     M=mean(ay.signals.values(:));
18     V=var(ay.signals.values(:));
19 xlabel({'Raw data of sensor ay' ['Mean: ' num2str(M)] ['Varianz: ' num2str
       (V)]}) % x-axis label
20 ylabel('Count of appearance') % y-axis label
21
22 %plot histogram of az and calculate meanvalue and variance
23 subplot(3,1,3)
24 hist(az.signals.values(:),50);
25     M=mean(az.signals.values(:));
26     V=var(az.signals.values(:));
27 xlabel({'Raw data of sensor az' ['Mean: ' num2str(M)] ['Varianz: ' num2str
       (V)]}) % x-axis label
28
29 %Magnetic sensor
30 figure(2)
31 %plot histogram of mx and calculate meanvalue and variance
32 subplot(3,1,1)
33 hist(mx.signals.values(:),50);
34     M=mean(mx.signals.values(:));
35     V=var(mx.signals.values(:));
36 title('Magnetic sensor')
37 disp(['Meanvalue: ' num2str(M)])
```

```
38 xlabel({'Raw data of sensor mx' ['Mean: ' num2str(M)] ['Varianz: ' num2str
       (V)]}) % x-axis label
39
40 %plot histogram of my and calculate meanvalue and variance
41 subplot(3,1,2)
42 hist(my.signals.values(:),50);
43     M=mean(my.signals.values(:));
44     V=var(my.signals.values(:));
45 xlabel({'Raw data of sensor my' ['Meanvalue: ' num2str(M)] ['Varianz: '
       num2str(V)]}) % x-axis label
46 ylabel('Count of appearance') % y-axis label
47
48 %plot histogram of mz and calculate meanvalue and variance
49 subplot(3,1,3)
50 hist(mz.signals.values(:),50);
51     M=mean(mz.signals.values(:));
52     V=var(mz.signals.values(:));
53 xlabel({'Raw data of sensor mz' ['Meanvalue: ' num2str(M)] ['Variance: '
       num2str(V)]}) % x-axis label
54
55 %Gyroscope sensor
56 figure(3)
57 %plot histogram of pitch and calculate meanvalue and variance
58 subplot(3,1,1)
59 hist(pitch.signals.values(:),50);
60     M=mean(pitch.signals.values(:));
61     V=var(pitch.signals.values(:));
62 title('Gyroscope sensor')
63 disp(['Meanvalue: ' num2str(M)])
64 xlabel({'Raw data of sensor pitch rate' ['Mean: ' num2str(M)] ['Varianz: '
        num2str(V)]}) % x-axis label
65
66 %plot histogram of roll and calculate meanvalue and variance
67 subplot(3,1,2)
68 hist(roll.signals.values(:),50);
69     M=mean(roll.signals.values(:));
70     V=var(roll.signals.values(:));
71 xlabel({'Raw data of sensor roll rate' ['Meanvalue: ' num2str(M)] ['
       Varianz: ' num2str(V)]}) % x-axis label
72 ylabel('Count of appearance') % y-axis label
73
74 %plot histogram of yaw and calculate meanvalue and variance
75 subplot(3,1,3)
76 hist(yaw.signals.values(:),50);
77     M=mean(yaw.signals.values(:));
78     V=var(yaw.signals.values(:));
79 xlabel({'Raw data of sensor yaw rate' ['Meanvalue: ' num2str(M)] ['
       Variance: ' num2str(V)]}) % x-axis label
80
81
82 %Complementary filter
83 figure(4)
```

```
subplot(3,1,1)
plot(Comp_pitch.time,Comp_pitch.signals.values(:));
title('Complementary filter')
xlabel('time [s]') % x-axis label
ylabel('pitch [degree]') % y-axis label

subplot(3,1,2)
plot(Comp_roll.time,Comp_roll.signals.values(:));
xlabel('time [s]') % x-axis label
ylabel('roll [degree]') % y-axis label

subplot(3,1,3)
plot(Comp_yaw.time,Comp_yaw.signals.values(:));
xlabel('time [s]') % x-axis label
ylabel('yaw [degree]') % y-axis label


%Kalman filter
figure(5)
subplot(3,1,1)
plot(Kalman_pitch.time,Kalman_pitch.signals.values(:));
title('Kalman filter')
xlabel('time [s]') % x-axis label
ylabel('pitch [degree]') % y-axis label

subplot(3,1,2)
plot(Kalman_roll.time,Kalman_roll.signals.values(:));
xlabel('time [s]') % x-axis label
ylabel('roll [degree]') % y-axis label

subplot(3,1,3)
plot(Kalman_yaw.time,Kalman_yaw.signals.values(:));
xlabel('time [s]') % x-axis label
ylabel('yaw [degree]') % y-axis label

%min max of magnetic sensor
maxz=max(mz.signals.values)*1000000;
minz=min(mz.signals.values)*1000000;
maxy=max(my.signals.values)*1000000;
miny=min(my.signals.values)*1000000;
maxx=max(mx.signals.values)*1000000;
minx=min(mx.signals.values)*1000000;

fprintf('\n');
disp('Calibration Data:');
disp('==================');
disp(['Min_X: ' num2str(minx)])
disp(['Max_X: ' num2str(maxx)])
disp(['Min_Y: ' num2str(miny)])
disp(['Max_Y: ' num2str(maxy)])
disp(['Min_Z: ' num2str(minz)])
disp(['Max_Z: ' num2str(maxz)])
```

```
136 fprintf('\n');
137
138 disp('====================================');
139 disp('Generating Calibration-Header-File');
140 disp('====================================');
141
142 disp(['Writing header ', headerFileName, ' ...' ]);
143 fileID = fopen(headerFileName,'w');
144 fprintf(fileID,'/*!\n');
145 fprintf(fileID,' * \\file OrientationDefines.h\n');
146 fprintf(fileID,' */\n');
147 fprintf(fileID,'\n');
148 fprintf(fileID,'#ifndef SIG_ORIENTATION_ORIENTATIONDEFINES_H_\n');
149 fprintf(fileID,'#define SIG_ORIENTATION_ORIENTATIONDEFINES_H_\n');
150 fprintf(fileID,'\n');
151 fprintf(fileID,'//Defines for Acceleration Magnetic angle calculation (
    automatic calibration via MATLAB)\n');
152 fprintf(fileID,'#define M_SIGORI_MAG_MINX_F64 %.9f\n',minx);
153 fprintf(fileID,'#define M_SIGORI_MAG_MAXX_F64 %.9f\n',maxx);
154 fprintf(fileID,'#define M_SIGORI_MAG_MINY_F64 %.9f\n',miny);
155 fprintf(fileID,'#define M_SIGORI_MAG_MAXY_F64 %.9f\n',maxy);
156 fprintf(fileID,'#define M_SIGORI_MAG_MINZ_F64 %.9f\n',minz);
157 fprintf(fileID,'#define M_SIGORI_MAG_MAXZ_F64 %.9f\n',maxz);
158 fprintf(fileID,'\n');
159 fprintf(fileID,'#endif /* SIG_ORIENTATION_ORIENTATIONDEFINES_H_ */\n');
160 fclose(fileID);
161 disp('Done');
162 fprintf('\n');
163 disp('Calibration successfully processed.');
```

Listing 12.1: MATLAB script to calibrate the magnetometer and automatically generate a header file with calibration data.

13 Remote Control Unit: PPM Decoding

13.1 Graupner PPM sum signal in a nutshell

The HElikopter Team of Hochschule Esslingen uses the Graupner radio remote control system to control the Quadrocopters. Therefore, it is necessary to decode the Graupner PPM Signal delivered by the radio receiver to give a user's input to the controller platform of the Quadrocopters.

The Graupner remote control is a multi-channel system that comprises the transmission of up to 12 channels. Each individual channel is encoded by a PWM signal with a periode time of 20ms (resulting in a update rate of 50Hz) as depicted in fig. 13.1. According to the transmitted value of each channel, the on-time T_{on} varies between 1ms (min. value) and 2ms (max. value).

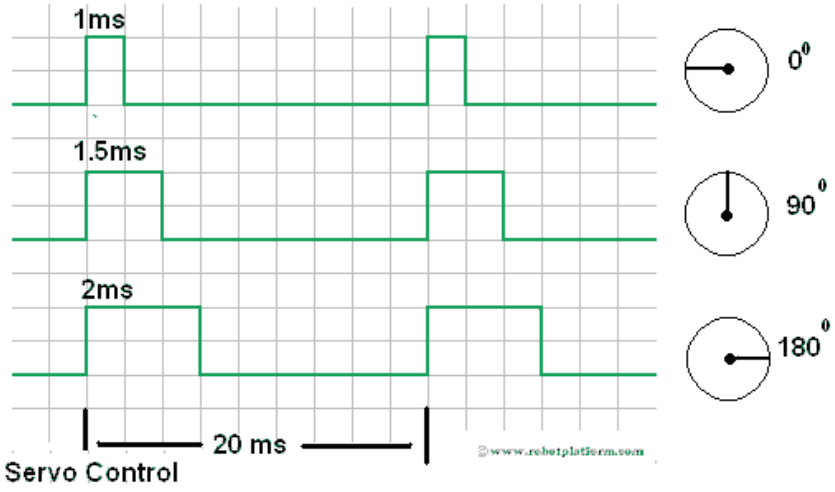


Figure 13.1: PWM signal to control a RC servo [RPL]

In order to transmit several (up to 12) PWM channels simultaneously, all channels are combined sequentially in one sum signal as depicted in fig. 13.2. This results in a PPM signal with a fixed pulse width. The time between the rising edges of two individual pulses encodes the on-time of the PWM signal of the corresponding channel. The remaining pause gap at the end of a PPM frame is used to resynchronize the transmission. Without this resynchronization, it would be impossible to map from a received pulse to the corresponding channel number.

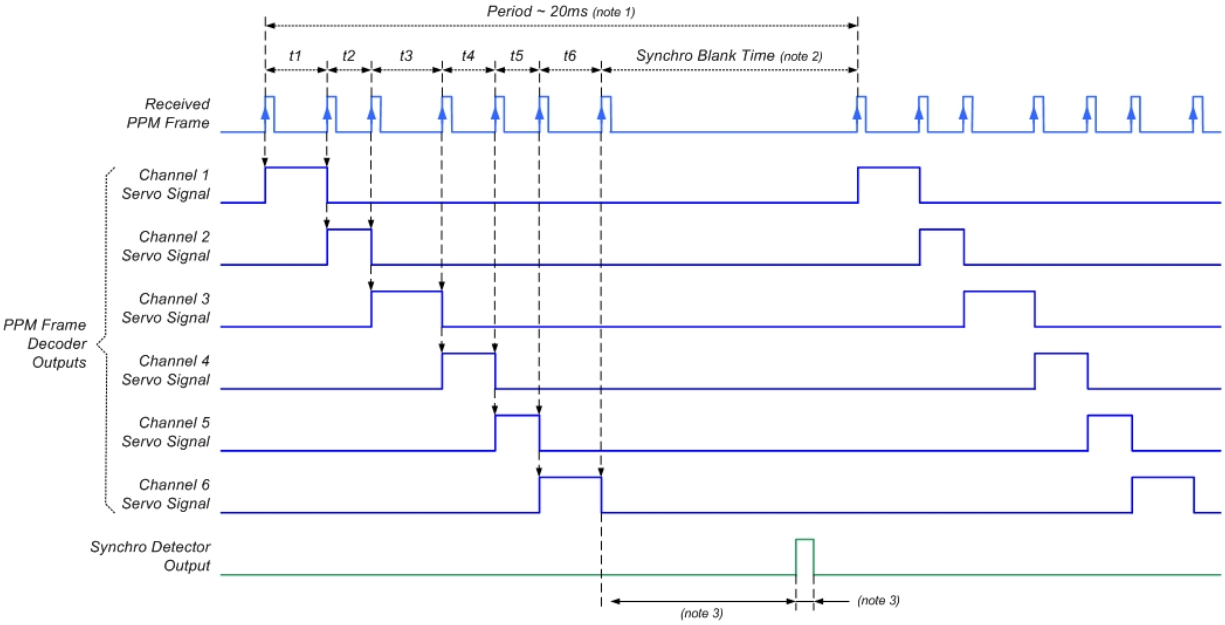


Figure 13.2: PPM sum signal scheme of an exemplary 6 channel remote control [REP]

Some Graupner systems have the same sum signal encoding, but with a inverted out signal (high-state becomes low-state, and vice-versa). But if triggered on rising or falling edges only, this fact will not harm the general encoding/decoding approach.

13.2 Building a custom kernel driver

To build a custom kernel driver it is mandatory to have all Linux kernel sources on your development machine. If this is not already the case, please see chapter 9 on how to get the latest kernel sources and how to build a custom kernel.

Furthermore, it is crucial that the kernel sources have the same version number as the kernel installed on the Raspberry Pi's firmware! Due to that fact, it is recommended to perform a kernel update and subsequently compile the custom kernel driver.

Below in listing 13.1, a generic `Makefile` is shown to cross-compile a custom kernel driver for the Raspberry Pi. The variable `KDIR` in line 7 specifies the path to the kernel sources. The variable `CROSS_COMPILE` in line 6 specifies the path to the raspberry pi cross-compile tools. Depending on the distribution and development environment, these paths may have to be adapted.

In line 11 of listing 13.1 the object file (with file extension `.o`) of the compilation process that shall be linked as a kernel module file (file extension `.ko`) gets defined. If another kernel driver file shall be compiled, put the c-code file name there and change the file extension to `.o`.

```
1  ARCH := arm
2  UNAME := $(shell uname -m)
3  ifeq ($(UNAME),armv6l)
4    KDIR := /usr/src/linux/
5  else
6    CROSS_COMPILE:=/home/user/rpi/tools/arm-bcm2708/gcc-linaro-arm-linux-
       gnueabihf-raspbian-x64/bin/arm-linux-gnueabihf-
7    KDIR := /home/user/rpi/linux/
8  endif
9
10 ifneq ($(KERNELRELEASE),)
11   obj-m := ppmDemux.o
12 else
13
14 default:
15   $(MAKE) ARCH=$(ARCH) CROSS_COMPILE=$(CROSS_COMPILE) -C $(KDIR) M=$(PWD)
       modules
16 endif
17
18 clean:
19   rm -rf *.ko *.o .*.cmd .tmp_versions Module.symvers
20   rm -rf modules.order *.mod.c
```

Listing 13.1: Simple Makefile to compile a custom kernel driver (here for the kernel module `ppmDemux`)

In the case of the `ppmDemux` kernel driver, a free running 64bit counter (called `System Timer`[1]) of the Raspberry Pi's System on Chip (SoC) has been used to determine a precise timing. This free running counter runs on $1Mhz$ independently of all other peripherals. This results in a maximum accuracy of $1\mu s$ of all measurements that utilize the `System Timer`.

The hardware address of the `System Timer` is defined by the base address `0x7E003000` plus the offset of the required data register (see [BCM], p. 175 ll.). Due to the `MMU` of the BCM2835 chip, the physical hardware address gets translated to the I/O base address `0x20003000` (as defined in line 27 of listing **??**). This address has to be used in the kernel module driver to access the free running $1Mhz$ `System Timer`.

Based on the articles [LMK70] and [LMK81], a kernel driver has been developed, to process a hardware trigger on GPIO pin 24. Whenever a rising edge is detected, a hardware interrupt is issued to the Kernel. The interrupt behavior in rising edges is defined in line 96 of listing **??**. The Interrupt Service Routine (ISR) is called in consequence (defined as function `hard_isr` in line 60 ll. of listing **??**).

```
60 static irqreturn_t hard_isr( int irq, void *dev_id )
61 {
62   /* read free running timer (64bit) @ 1MHz */
63   modIrqObj.timer_l = readl(timer_low);
```

1 See the peripherals datasheet on Raspberry Pi's SoC Broadcom BCM2835, p. 172 ll.

```
64   modIrqObj.timer_h = readl(timer_high);
65
66   /* counts rising edges since last read */
67   modIrqObj.irqNum += 1;
68
69   return IRQ_WAKE_THREAD;
70 }
```

This Interrupt Service Routine is the first function called whenever a rising edge event on GPIO24 occurs. The value of the 64bit `System Timer` get read out and an interrupt counter gets increased. That way, even for the unprobable case that one interrupt gets unprocessed, the synchronization of the Graupner PPM sum signal can be guaranteed.

After the hard ISR[1] has been processed, the threaded interrupt handler `rpi_gpio_isr` (line 52 ll. of listing **??**) gets called. For complex kernel drivers, in this function can be implemented complex algorithms to process the captured signals. This function is threaded and can be prioritized by the kernel's scheduler to guarantee a minimal latency time of the system.

```
60 static irqreturn_t hard_isr( int irq, void *dev_id )
61 {
62   /* read free running timer (64bit) @ 1MHz */
63   modIrqObj.timer_l = readl(timer_low);
64   modIrqObj.timer_h = readl(timer_high);
65
66   /* counts rising edges since last read */
67   modIrqObj.irqNum += 1;
68
69   return IRQ_WAKE_THREAD;
70 }
```

When all statements of the soft (threaded) ISR `rpi_gpio_isr` has been processed, the read function of the kernel driver get waken up (see listing **??** in line 55 and line 118 for the counterpart in the read function).

```
52 static irqreturn_t rpi_gpio_isr( int irq, void *data )
53 {
54   /* wake up read function to retrieve new data */
55   wake_up( &sleeping_for_ir );
```

```
117  /* wait for an rising edge event (caught by ISR) */
118  wait_event_interruptible( sleeping_for_ir, modIrqObj.irqNum);
```

1 Code inside of the hard ISR should be reduced to the absolute minimum and only necessary tasks should be performed. Every unnecessary CPU cycle in this section will slow down the system performance dramatically!

The complete code of the `ppmDemux` kernel module driver can be seen in listing **??** and in the SVN repository in `/impl/trunk/kern/`.

In order to compile the kernel driver, navigate to folder where the c code files and the `Makefile` are located and execute the following command in the bash command line:

```
cd ../path/to/kernelDriverFiles/
make
```

This will produce the compile kernel module object file `ppmDemux.ko`. If the kernel module has been cross-compiled on the Development Environment (VM) then copy this `.ko`-file to the Raspberry Pi's file system by using scp.

```
scp ./ppmDemux.ko pi@192.168.2.1:~/
```

In the above shown command, the kernel module object file will be copied to the home directory of the user `pi` on the Raspberry Pi's Linux system. If the IPv4 address has been changed, this has to be adapted when using the above stated command (here used: standard IP address of pre-configured Raspberry Pi Firmware).

After the file transfer is finished, the kernel module can be loaded to the Linux System. Navigate to the location of the stored kernel module object file (in this example the home directory of the user `pi`) and use the command `insmod`.

```
cd /home/pi/
sudo insmod ppmDemux.ko
```

If the kernel module is successfully loaded (no error messages occurred), then a character file handler `/dev/gpioirq24` has been created by the kernel module driver `ppmDemux`. Get noticed of rising edges on GPIO pin 24, read the character file handler `/dev/gpioirq24` as shown in the exemplary test code as shown in listing **??** or file `getPpmDemuxValues.c` in the SVN repository folder `/impl/trunk/kern/tst/`.

13.3 Stimulating the GPIO-Pins for validation

For testing and validation purposes of the custom built Kernel Driver `ppmDemux` of chapter 13.2, it was required to stimulate the GPIO pin 24 with a PPM signal similar to a Graupner sum signal.

For this purpose, a STM32F4 Discovery board was used to run a specifically written firmware that produces a variable PPM signal as depicted in fig. 13.3. The firmware of the STM32F4 board is capable to produce PPM signals with a pulse-to-pulse with of $480\mu s$ up to $2560\mu s$. By pressing the `blue button` of the STM32F4 Discovery board as

shown in fig. 13.3, the pulse-to-pulse width can be altered in the sequence as shown on table 13.1.

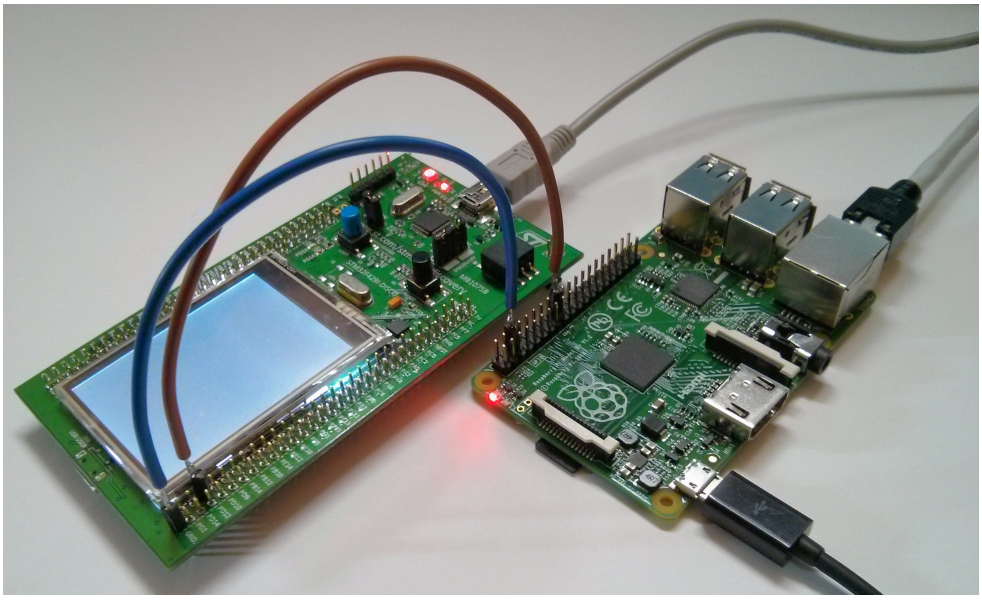


Figure 13.3: In order to test the PPM kernel driver, a STM32F4 Discovery Board was used to stimulate the GPIO Pin 24 of Raspberry Pi Board. The signals got also observed by a 50 MHz Oscilloscope to validate the measured PPM length.

The firmware developed for the STM32F4 Discovery board can be found in the SVN repository in `/impl/trunk/kern/tst/STM32F4_PWM_Stimulator`. To use the firmware, first the CooCox IDE[1] has to be installed on the development machine as well as the GCC Toolchain for ARM Embedded Processors[2]. Then the provided project file in the SVN repository can be opened and customized, if needed.

1 http://www.coocox.org/software/coide.php
2 https://launchpad.net/gcc-arm-embedded/+download

State No.	Pulse-to-pulse witdth	State No.	Pulse-to-pulse witdth
1	$480\mu s$	10	$1590\mu s$
2	$620\mu s$	11	$1710\mu s$
3	$730\mu s$	12	$1840\mu s$
4	$860\mu s$	13	$1960\mu s$
5	$970\mu s$	14	$2080\mu s$
6	$1100\mu s$	15	$2200\mu s$
7	$1220\mu s$	16	$2320\mu s$
8	$1350\mu s$	17	$2440\mu s$
9	$1460\mu s$	18	$2560\mu s$

Table 13.1: Pulse-to-pulse width in microseconds for each state of the STM32F4 firmware

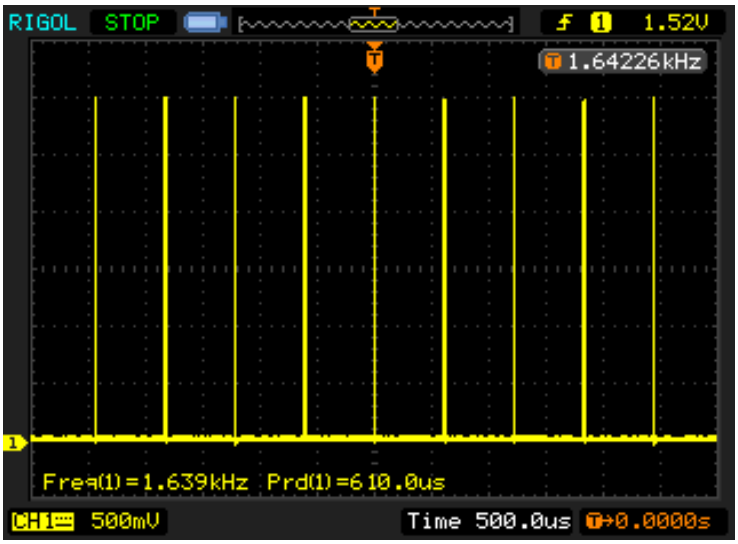


Figure 13.4: Oscilloscope graph of stimulus signal emitted by the STM32F4 Discovery board (Custom STM32F4 firmware for $610\mu s$ pulse-to-pulse width).

To validate the stimulus signal output of the STM32F4 Discovery board, the output signals where observed with a 50Mhz oscilloscope. Screenshots of two PPM signals are shown in fig. 13.4 and fig. 13.5.

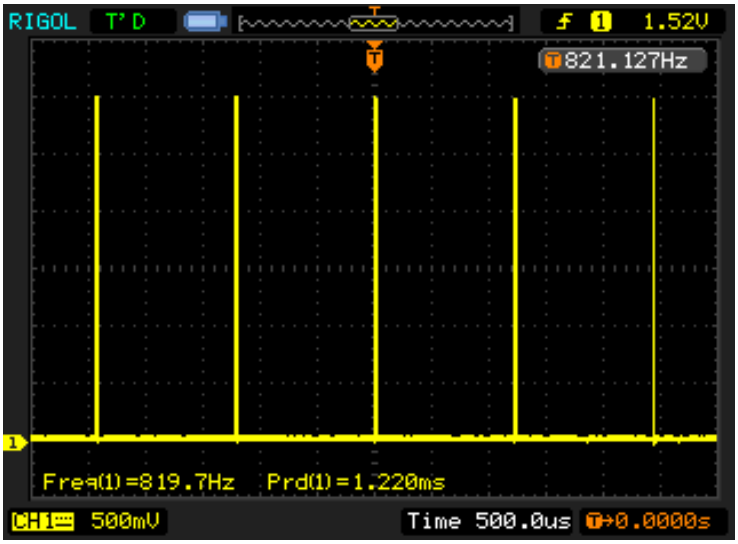


Figure 13.5: Oscilloscope graph of stimulus signal emitted by the STM32F4 Discovery board (Custom STM32F4 firmware for 1220μs pulse-to-pulse width).

13.4 Results on experimental kernel driver

Before the custom kernel module driver `ppmDemux` has been developed by the authors of this document, a performance test of the standard Kernel's GPIO interface has been evaluated. For this purpose, a 1220μs PPM signal was generated (as shown in chapter 13.3) and measured with a simple test script.

The measured pulse-to-pulse time was measured under system idle and system under full load. The 'system under full load' scenario was produced by executing 3 threads that run at full load as well as 3 threads that produce full I/O-load. To achieve this, the program `stress` was installed on the Raspberry Pi's Linux system by issuing the command shown below on the Raspberry Pi's bash command line

```
sudo apt-get install stress
```

To set the test system under a full load scenario, the command `stress` got issued as shown below:

```
sudo stress -c 3 -i 3 &
```

This command starts 3 threads that each consume the max. CPU cycles available[1]. Additonally 3 threads are started that each put a max. I/O load on the system. This setup can be considered as a maximum load scenario for a the single core Raspberry Pi B+ board.

For each scenario (idle and full load), approximately 10000 measurements where taken and stored. In fig. 13.6 and fig. 13.7 the results of these measurements can be seen.

When using the standard Kernel's GPIO interface, the accuracy of the measurement under system idle is still in a acceptable range although latency jitter exceeds the acceptable range in rare cases (compare with fig. 13.6).

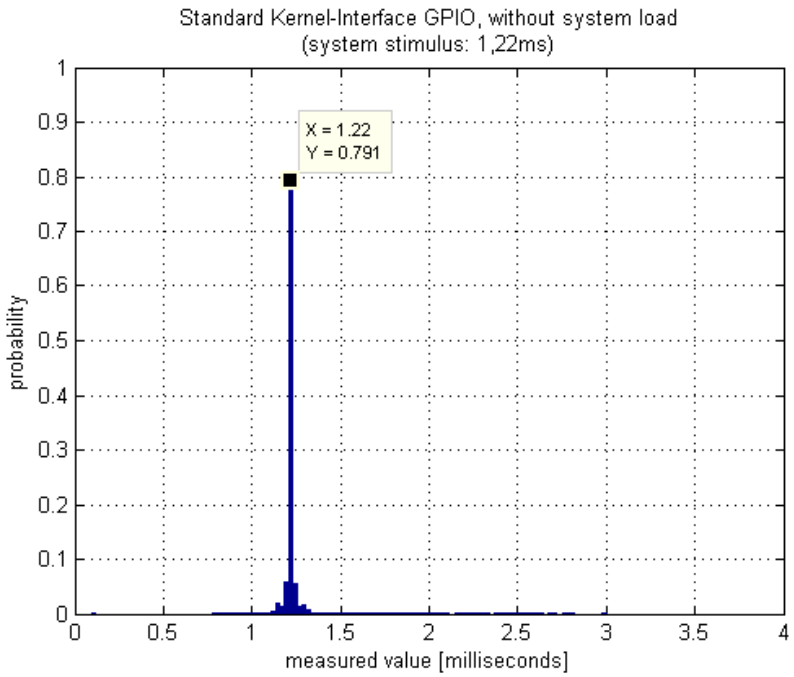


Figure 13.6: Measured pulse period time with standard kernel's GPIO interface (system on idle)

On the other hand, when the system is under full load the latency jitter exceeds the acceptable range by far. As in fig. 13.7 can be seen, approximately 25% of all measured pulse-to-pulse widths are missed. This results in measured pulse-to-pulse width that is double the width of the stimulus signal. The second spike at $2.5ms$ indicates this effect. Unfortunately, for a mechanism to measure the user's remote control input, this is a unacceptable situation.

Therefore, the development of a custom kernel module driver called `ppmDemux` has been

1 The ampersand character (`&`) at the end of the line runs the processes in the background of the system. To stop the program `stress` type the command '`sudo killall stress`' to the bash command line. This will terminate all processes of the program `stress` and set the system back to an idle state.

started. The proof-of-concept approach of chapter 13.2 has been tested under a Preempt_RT patched Kernel environment, as shown in chapter 9.2.

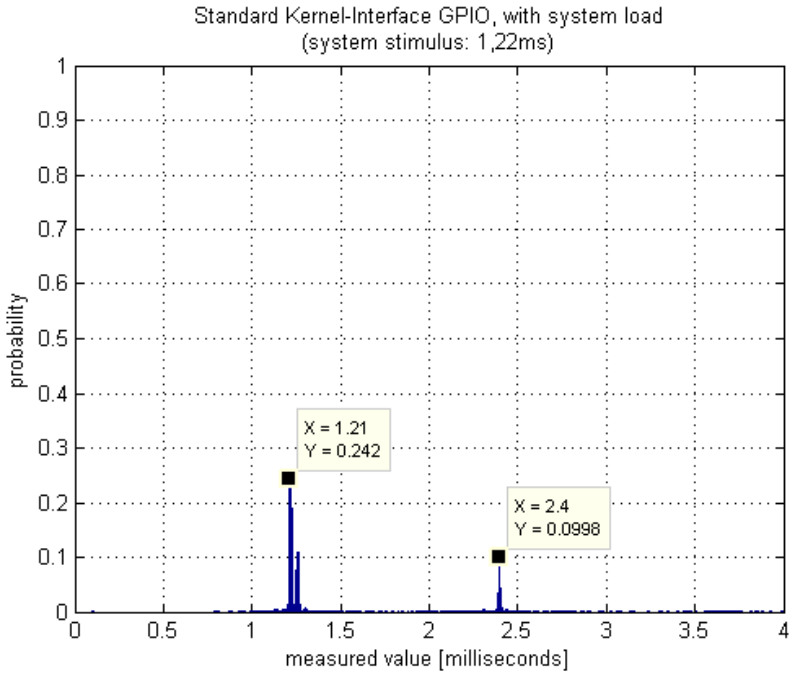


Figure 13.7: Measured pulse period time with standard kernel's GPIO interface (system under full load). The second spike at 2.4ms shows that many rising edges of the stimulus signal are missed. This is a non-acceptable result for a reliable system.

The custom kernel driver `ppmDemux` has been test again under the two test scenarios: system idle and system under full load (using the command `stress`). Under system idle conditions, fortunately the measured results are very good. The latency is negligible small and the latency jitter has a variance of roughly $10\mu s$ for a stimulus PPM signal with a pulse-to-pulse width of $1220\mu s$ (as shown in fig. 13.8). With some post-processing filters, a measurement accuracy of 1%-3% is a very realistic range for a future implementation.

Fortunately, even under heavy system load (full load scenario), the latency is again negligible. Also the latency jitter rises only marginally. A variance of not more than $53\mu s$ for a stimulus PPM signal under full load can be assumed (as depicted in fig. 13.9). No single edge of the PPM signal has been missed. Since the measurement shows a robust and reliable measurement behavior of the custom kernel driver, this approach is considered to be acceptable for the high reliability requirements of this project.

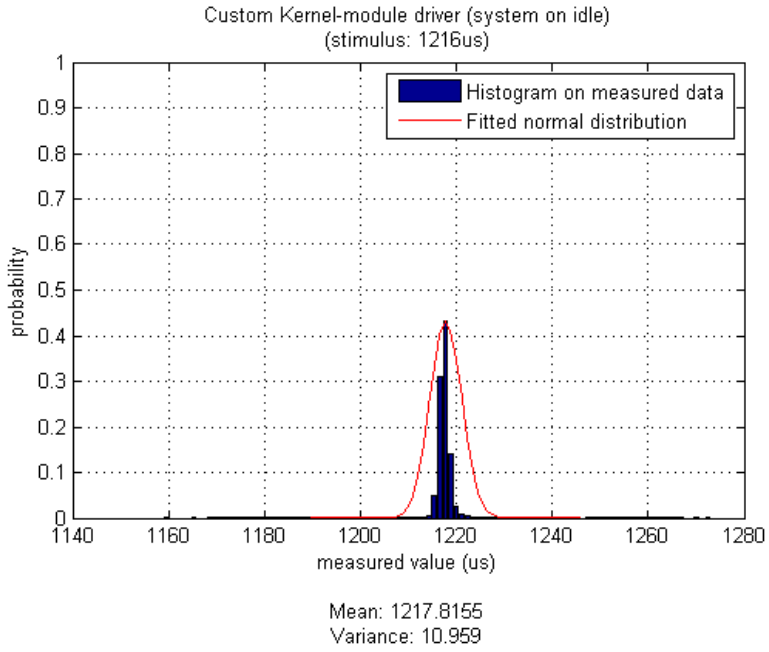


Figure 13.8: Measured pulse period time with `ppmDemux` (system on idle). The very low latency jitter shows the superior performance of the kernel driver approach.

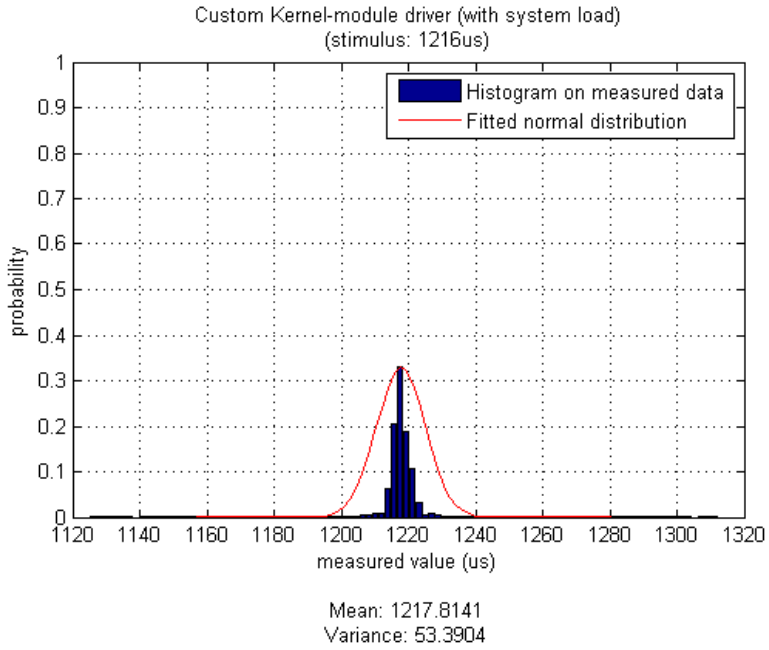


Figure 13.9: Measured pulse period time with `ppmDemux` (system under full load). Even under system's full load, the latency jitter is in an acceptable range. No single rising edge of the stimulus signal is missed. This shows that the kernel driver approach is the best choice for the high reliability requirements of this project.

14 Defined Tests for our sensors

14.1 Infrared Distance Sensor

For detailed informations see also 16 or Data sheet.

Infrared Ranging Sensor, intended to measure the distance to ground.

We tested our Infrared Sensor with three different distances: 16cm, 49cm and to the ceiling (which represent the maximum of 150cm). Gathering a lot of data (10 000 values) for each measurement gets a an impression of the jitter and the precision of the value.

In the second steps we did the measurement with different surfaces. They were changing that much, that we search for another possibility to measure the distance. See xxxxxx

14.2 Ultrasonic Distance Sensor

These tests were only done in theory. Before we finally decided to proceed with that concept, we stoped it.

As we knew that the use of one ultrasonic sensor gives us not that good data and depening of the horizontal angel of the HElikopter, we thought that using three ultrasonic sensors could give us better distance data, independent from the flight angel. But merging 3 sensor to get one good value would a lot of effort and still don't provide us that good data.

While thinking about that concept, we found a better alternative.

14.3 LIDAR-lite Laser Distance Sensor

For detailed information about the Sensor see also the datasheet or Chapter 15.5.

We choose several distances, to validate the measurement. Always taking at least 10 values, to get an impression of the jitter. As the sensor is very good, there is only very small jitter above 20cm.

As the sensor supports distance up to 40 meters, we only tested it with 4 certain distances:
very small distance: under 1 meter (0,69m/0,20m)
small distance: between 1 and 2 meters (1,46/1,68m)
middle distance: above 3 meters ()
high distance: above 10 meters ()

Even the measurements diagonal through a room, with a big impact angle, worked really good.
Of course we also tested the sensor to different surfaces.

14.4 Inertial Measurement Unit (IMU)

The IMU consists of these 3 sensors:

14.4.1 Acceleration and Magnet Sensor(Compass)

The easiest way to test is, is manually. You trace, gather and display the data from the sensor and do some movements. You can move it as fast as possible over different distances. It should be possible to see at the different curves, how/weather the sensor works.

The Magnet sensor should be tested in a similar way. Depending on the magnetic fields in building, you can display the data and observe how the values change, when you turn the sensor/board/quadrocopter in different directions. In this case it is only relevant in which speed and with which divergence the sensor shows the data. The sensor delivers an absolute value, in which direction your sensor is looking at.

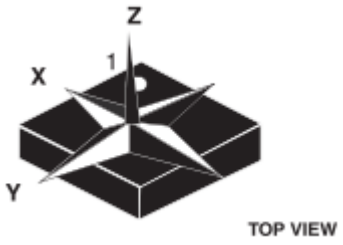


Figure 14.1: Compass, Source: Data sheet

14.4.2 Gyroscope Sensor

This sensor gives information about the angles of all axis, how the quadrocopter lays in the air.
We check the work of sensor with observing the live data, as before.

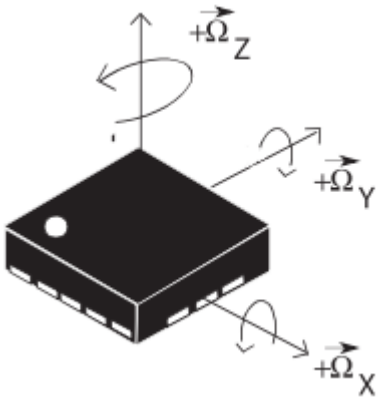


Figure 14.2: Gyroscope, Source: Data sheet

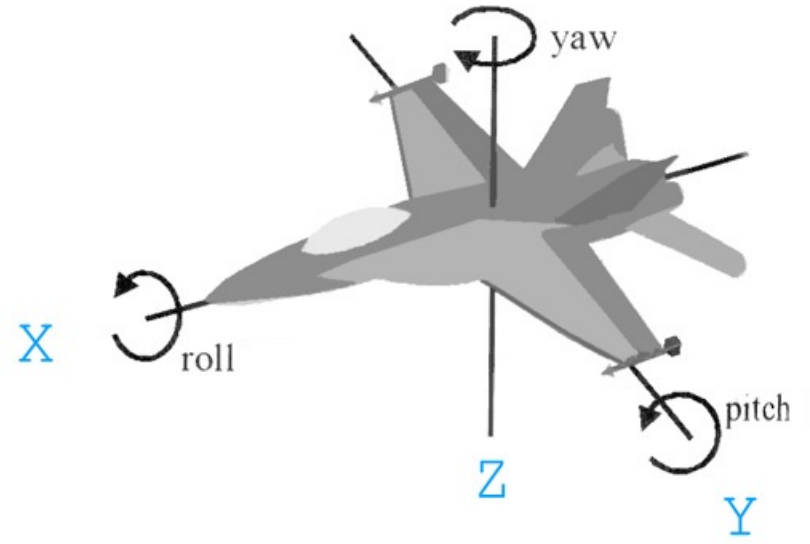


Figure 14.3: Example at plane, Source: timzaman.com

14.4.3 Pressure Sensor

This sensor delivers a height compared to the sea level. You can say it is a absolute value. For that reason it easy to place the sensor at certain heights, and took a lot of data. So we can see the jitter. When lifting it up/down fast for several meters, we can check how fast we get the updated data.

In future this sensor will be controlled with the data form the distance measurements to the ground under us. So we can eliminate the influences from the weather for example.

15 Sensors and limits

This chapter is about the sensors we use and their maximum and minimum borders. Compared with the physical needed values, we try to realize a autonomous flight.

The sensors will be connected to a Raspberry Pi Board mounted on a Quadrocopter (HElicopter).

15.1 Analog- Digital Converter

Used Model: Adafruit ADS1015

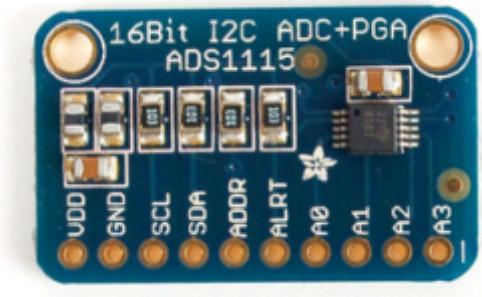


Figure 15.1: Example of ADC, ADS1115

Resolution: 12 Bits
Programmable Sample Rate: 128 to 3300 Samples/Second
Power Supply/Logic Levels: 2.0V to 5.5V
Low Current Consumption: Continuous Mode: Only 150µA Single-Shot Mode: Auto Shut-Down
Internal Low-Drift Voltage Reference
Internal Oscillator
Internal PGA: up to x16
I2C Interface: 4-Pin-Selectable Addresses
Four Single-Ended or 2 Differential Inputs
Programmable Comparator

15.2 Infrared Analog Distance Sensor

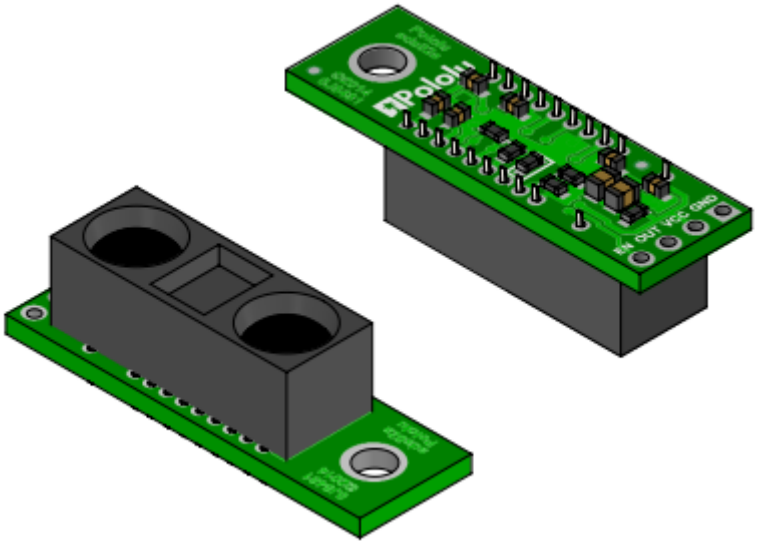


Figure 15.2: IR Sensor on Carrier Board

Mounted IR Modul: Sharp GP2Y0A60SZLF

1. Distance measuring sensor is united with PSD, infrared LED and signal processing circuit.
2. Distance measuring range : 10 to 150 cm
3. Compact size (22.0 × 8.0 × 7.2mm)
4. Long distance measuring type (No external control signal required)
5. Analog output type
6. Update time: 16.5ms ± 3.7ms

15.3 GPS

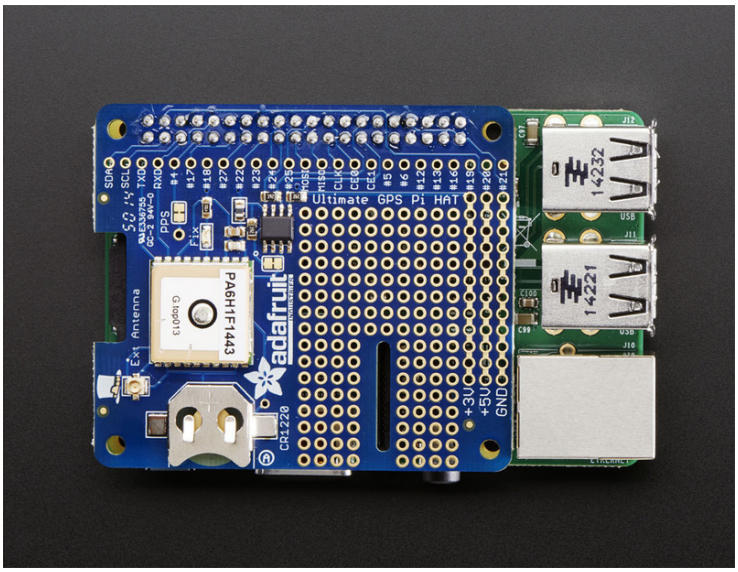


Figure 15.3: Adafruit GPS Hat

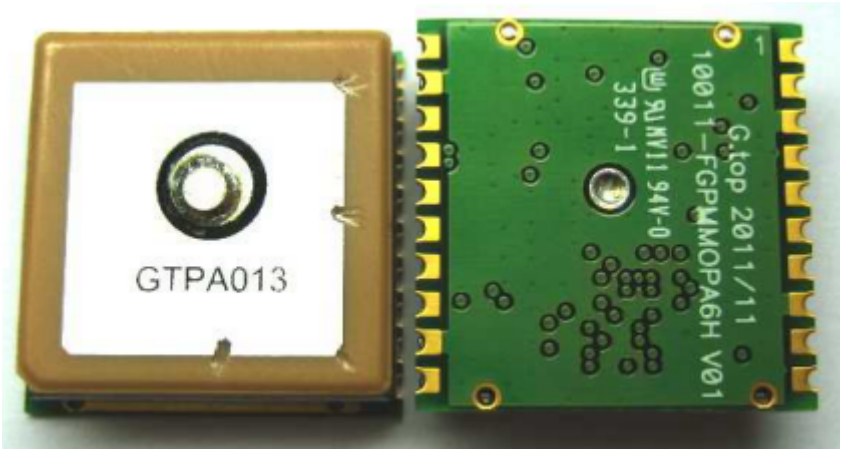


Figure 15.4: GPS modul

- Built-in 15x15x2.5mm ceramic patch antenna on the top of module
- Ultra-High Sensitivity: -165dBm (w/o patch antenna), up to 45dB C/N of SVs in open sky reception.
- High Update Rate: up to 10Hz (note1)
- 12 multi-tone active interference canceller (note2)
- High accuracy 1-PPS timing support for Timing Applications (10ns jitter)
- AGPS Support for Fast TTFF ("'EPO"' Enable 7 days/14 days)
- Self-Generated Orbit Prediction for instant positioning fix

- "‘AlwaysLocate"’ (note2)
- Intelligent Algorithm (Advance Power Periodic Mode) for power saving
- Logger function Embedded (note2)
- Automatic antenna switching function
- Antenna Advisor function
- Gtop Firmware Customization Services
- Consumption current(@3.3V):
Acquisition: 25mA Typical
Tracking: 20mA Typical
- E911, RoHS, REACH compliant

note 1: SBAS can only be enabled when update rate is less than or equal to 5Hz.
note2: Some features need special firmware or command programmed by customer, please refer to G-top "‘GPS command List"’

15.4 Inertia Measurement Unit (IMU)

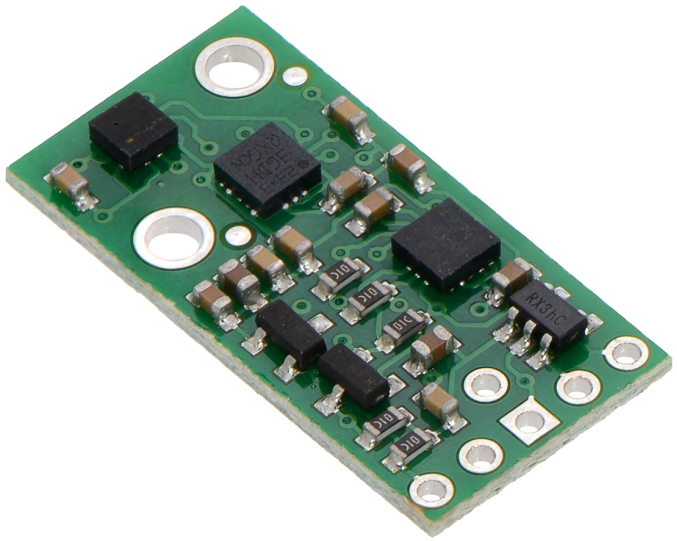


Figure 15.5: Inertia Measurement Unit

Model: Pololu AltIMU

15.4.1 3D accelerometer and 3D magnetometer module

Features
- 3 magnetic field channels and 3 acceleration channels
- ±2/±4/±8/±12 gauss dynamically selectable magnetic full-scale
- ±2/±4/±6/±8/±16 g dynamically selectable linear acceleration full-scale
- 16-bit data output
- SPI / I2C serial interfaces
- Analog supply voltage 2.16 V to 3.6 V
- Power-down mode / low-power mode
- Programmable interrupt generators for free- fall, motion detection and magnetic field detection
- Embedded temperature sensor
- Embedded FIFO

Working of an accelerometer: The accelerometer is a small MEMS, that works on the principle of force acting on the MEMS. The micro springs attached to a piezoelectric component induces electricity in a capacitor. The output voltage is the measure of how strong the microsprings were pushed or pulled. One can think of an elevator where the mass reduces when the elevator is going down and hence the person inside the elevator can feel that the elevator is accelerating downwards. Hence, the basic principle of operation behind the MEMS accelerometer is the displacement of a small proof mass etched into the silicon surface of the integrated circuit and suspended by small beams.

15.4.2 MEMS pressure sensor

Features
- 260 to 1260 mbar absolute pressure range
- High-resolution mode: 0.020 mbar RMS
- Low power consumption:
- Low resolution mode: 5.5 uA
- High resolution mode: 30 uA
- High overpressure capability: 20x full scale
- Embedded temperature compensation
- Embedded 24-bit ADC
- Selectable ODR from 1 Hz to 25 Hz
- SPI and I2C interfaces
- Supply voltage: 1.71 to 3.6 V
- High shock survivability: 10,000 g
- Small and thin package
lead-free compliant

15.4.3 MEMS motion sensor: three-axis digital output gyroscope

Features
- Wide supply voltage, 2.2 V to 3.6 V
- Wide extended operating temperature range
(from -40 °C to 85 °C)
- Low voltage compatible IOs, 1.8 V
- Low power consumption
- Embedded power-down
- Sleep mode
- Fast turn-on and wake-up
- Three selectable full scales up to 2000 dps
- 16 bit rate value data output
- 8 bit temperature data output
- I2C/SPI digital output interface
- 2 dedicated lines (1 interrupt, 1 data ready)
- User enable integrated high-pass filters
- Embedded temperature sensor
- Embedded 32 levels of 16 bit data output FIFO
- High shock survivability

15.5 LIDAR-Lite Laser Ranging Module

Figure 15.6: PULSEDLIGHT LIDAR Laser Sensor

Model LL-905-PIN-01

Performance
Range: 0-20m LED Emitter
Range: 0-60m Laser Emitterv (at full sunlight: 40m)
Accuracy: +/- 0.025m
Power: 5vdc, <100ma
Acquisition Time: < 0.02 sec
Rep Rate: 1-100Hz

Spread
At very close distances (less than a meter) the beam is about the size of the aperture (lens), at distances longer than that you can estimate it using this equation:

Distance/100 = beam size at that distance (in whatever units you measured distance in).

The actual spread is 8 milli-radians or 1/2 degree.

Configurations

- LED/PIN Diode, No Optics
- LED/PIN Diode, 12mm Optics
- Laser/PIN Diode 14mm Optics
(Class 1 Laser Product)

Interface
- I2C
- PWM

General Technical Specifications

Power 4.75 - 5.5V DC Nominal, Maximum 6V DC
Weight PCB 4.5 grams, Module 22 grams with optics and housing
Size PCB 44.5 X 16.5mm (1.75" by .65")
Housing 20 X 48 X 40mm (.8" X 1.9" X 1.6")
Current Consumption <2mA @ 1Hz (shutdown between measurements), <100mA (continuous operation)
Max Operating Temp. 70° C
External Trigger 3.3V logic, high-low edge triggered
PWM Range Output PWM (Pulse Width Modulation) signal proportional to range, 1msecmeter, 10µsec step size
I2C Machine Interface 100Kb - Fixed, 0xC4 slave address. Internal register access & control.
Supported I2C Commands Single distance measurement, velocity, signal strength
Mode Control Busy status using I2C, External Trigger input PWM outputs
Max Range under typical conditions aprox. 40m
Accuracy +/- 2.5cm, or +/- 1"
Default Rep Rate aprox. 50 Hz.

Laser Parameters

Wavelength: 905 nm (nominal)
Total Laser Power - Peak: 1.3 Watts
Mode of operation: Pulse (max pulse train 256 pulses)
Pulse Width: 0.5 uSec (50% duty Cycle)
Pulse Repetition Frequency: 10-20 KHz nominal
Energy per Pulse: <280 nJ
Beam Diameter at laser aperture: 12 mm x 2 mm
Divergence: 4 m Radian x 2 m Radian (Approx)

Innovation Summary

- The use of a signature matching technique (known as signal correlation) that

estimates time delay by electronically sliding a stored transmit reference over the received signal in order to find the best match.

- Operation of the infrared LED or laser in short bursts allowing a 100:1 advantage in peak output power over measurement systems using a continuous beam.
- Decreased measurement times down to a millisecond or less allows significant power consumption advantages and high repetition rates for scanning applications.
- We have developed novel current driver technology with nanosecond signal transition times at high peak currents to produce high power transmit burst sequences.
- Our signal processing approach is implementable in a single programmable logic chip or System-on-Chip (SoC) to allow deployment without the costly development of custom processing chips.
- Detector switching technology allows multiple detectors to be processed by a single signal-processing channel. Enabling compact multichannel systems deployable in under one square inch of board space.
- Multiple digital processing cores implementable in a single cost effective programmable logic chip.
- Optical scanner technology to multiply low-resolution electronic scanning to higher resolutions.

Source: Data sheet

15.5.1 Technology

PulsedLight‘s "Time-of-flight" distance measurement technology is based on the precise measurement of the time delay between the transmission of an optical signal and its reception. The patented, high accuracy measurement technique enables distance measurement accuracy down to 1 cm by the digitization and averaging of two signals; a reference signal fed from the transmitter prior to the distance measurement and a received signal reflected from the target. The time delay between these two stored signals is estimated through a signal processing approach known as correlation, which effectively provides a signature match between these two closely related signals. The correlation algorithm accurately calculates the time delay, which is translated into distance based on the known speed-of-light. A benefit of PulsedLight‘s approach is the efficient averaging of low-level signals enabling the use of relatively low power optical sources, such as LEDs or VCSEL (Vertical-Cavity Surface-Emitting) lasers, for shorter-range applications and increased range capability when using high power optical sources such as pulsed laser diodes.

Source: LidarLite Operating Manual

15.5.2 possible measurement problems

When the LIDAR-Lite unit sometime unexpected results, first think about these points:

There are several variables to consider if your LIDAR-Lite fails to return a valid measurement or seems not to recognize an object at all. These can be categorized into the following areas:

A. Reflectivity of the object
B. Distance of the object from the sensor
C. Size of the object relative to the transmitted infrared beam
D. Direct or reflected sunlight finding its way into the receiver
E. Atmospheric conditions
F. Obstruction of the receiver lens
F. Failure of the LIDAR-Lite unit

We'll consider reflectivity here:

Reflectivity

Reflective characteristics of an object's surface can be divided into three categories (in the real world, a combination of characteristics is typically present):
A. Diffuse Reflective
B. Specular, and
C. Retro-reflective

Diffuse Reflective

In the case of purely diffuse surfaces, we are talking about materials that have a textured quality that causes reflected energy to disperse uniformly. This tendency results in a relatively predictable percentage of the dispersed laser energy finding its way back to the LIDAR-Lite receiver. As a result, these materials tend to read very well. Materials that fall into this category are paper, matte walls, and granite. It is important to note that materials that fit into this category due to observed reflection at visible light wavelengths may exhibit unexpected results in other wavelengths. The near infrared range used by the LIDAR-Lite transmitter may detect them as nearly identical. A case in point is a black sheet of paper may reflect a nearly identical percentage of the infrared signal back to the receiver as a white sheet.

Specular

Specular surfaces, on the other hand, are difficult or impossible for the LIDAR-Lite to recognize because radiated energy is not dispersed. Reflections off of specular surfaces tend to reflect with little dispersion which causes the reflected beam to remain small and, if not reflected directly back to the receiver, to miss the receiver altogether. The LIDAR-Lite may fail to detect a specular object in front of it unless viewed from the normal. Examples of specular surfaces are mirrors and glass viewed off-axis.

Retro-reflective

Retro-reflective surfaces return a very high percentage of radiated energy to the receiver due to their reflective properties. Light hitting a retro-reflective surface will return to the receiver without much signal loss so retro-reflective surfaces are typically very good targets for the LIDAR-Lite. Paint used to mark roadways, animals' eyes, license plates and road signs are examples of retro-reflective surfaces. Some bicycle reflectors are retro-reflective in the visible spectrum but are not easily detected by LIDAR-Lite due, in part, to their failure to reflect infrared wavelengths as efficiently as they do light in the visible spectrum.

16 Infrared Sensor

To achieve our main goal, the autonomous landing, we tried several different sensor. In this Chapter we introduce them, including the problems we were faced and the configurations we did.

We tried to establish a distance measurement with GP2Y0A60SZLF Infrared Sensor on a Pololu Carrier Board. The Measuring distance of that sensor is 10 to 150 cm. For detailed technical values, please see the sensor chapter or the datasheet.

16.1 First steps

Our first action was to check, weather the sensor is working or not. So we need to active the I²C Bus on the Raspberry Pi and set up the Analog Distance Converter (ADS1011), because the Infrared Sensor provides us only a analog output.
We did a fast test of the sensor using python, just for checking the sensor is not broken.

To gather a bunch of data, we developed I²C and ADC Driver ourselves in C.

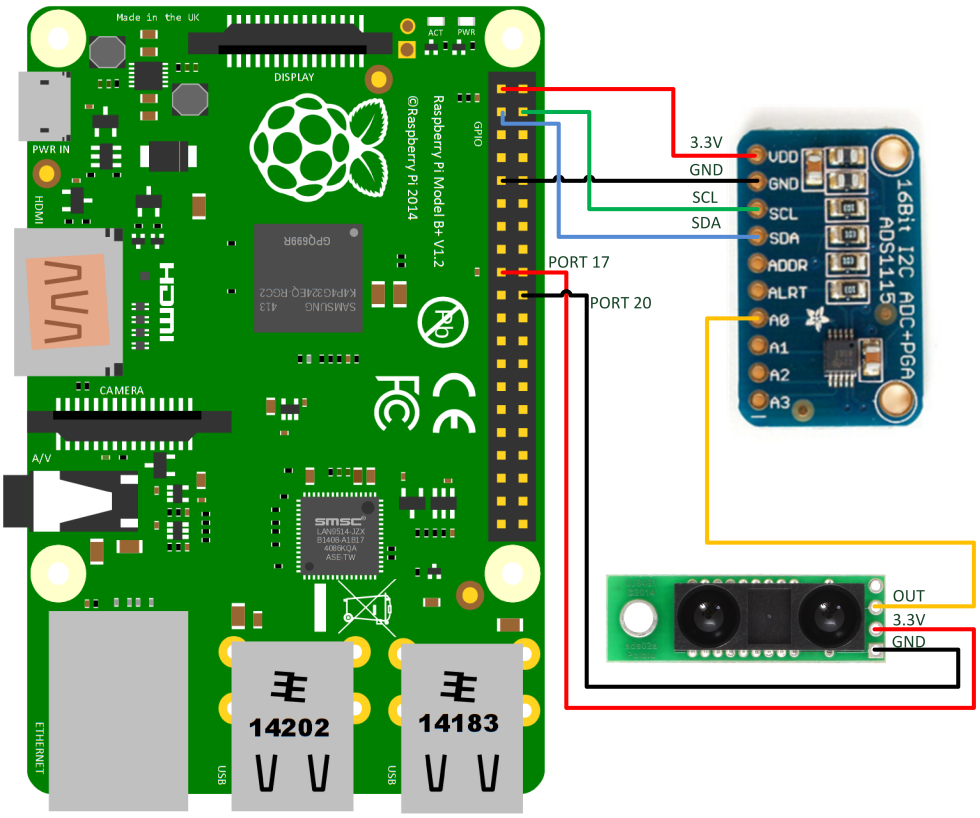


Figure 16.1: Wiring ADC and IR

16.1.1 ADC Configuration

First Hex depends on Starting Conversion + the Input, which Pin to read A0-3,
Second Value is PGA (001)=+-4,099V and continuous Mode (0).

These three bytes are written to the ADS1015 to set the config register and start the conversion.

l_writeBuf_rg24[0] = 1;
This sets the pointer register to write two bytes to the config register

l_writeBuf_rg24[1] = l_mux_ui8;
This sets the 8 MSBs of the config register (bits 15-8) to 11000011

l_writeBuf_rg24[2] = 0x23;
This sets the 8 LSBs of the config register (bits 7-0) to 00100011

The following table shows the Hex values in the direction from top to bottom what is needed for reading a conversion value on the specific inputs. At the empty field, there is no change compared to Input A0.

	Input A0	Input A1	Input A2	Input A3
Slave adress+RW	0x49			
PTR register	0x01			
MSB Config	0xC2	0xD2	0xE2	0xF2
LSB Config	0x23			
Slave adress+RW	0x49			
PTR register	0x00			
Data from Slave	0xXX	0xXX	0xXX	0xXX
Data from Slave	0xXX	0xXX	0xXX	0xXX

Table 16.1: ADC Conversion Read

MSB:
The first hexadecimal value is to start the conversion and depends on the Input, which Pin to read A0-3.
The second hexadecimal value is PGA (001)= +-4,099V and continuous Mode (0).

LSB:
The first hexadecimal value is the sample Rate. (001) sets it to 250SPS + Comp Mode (0).
The second hexadecimal value is the Comp. config. (0011) disables the comparator.

16.2 Measured Values

With our own logic getting the data from the sensor, we could store as much data we want. Now we were able to get a impression on the jitter and could thing about a good algorithm to get reliable values.

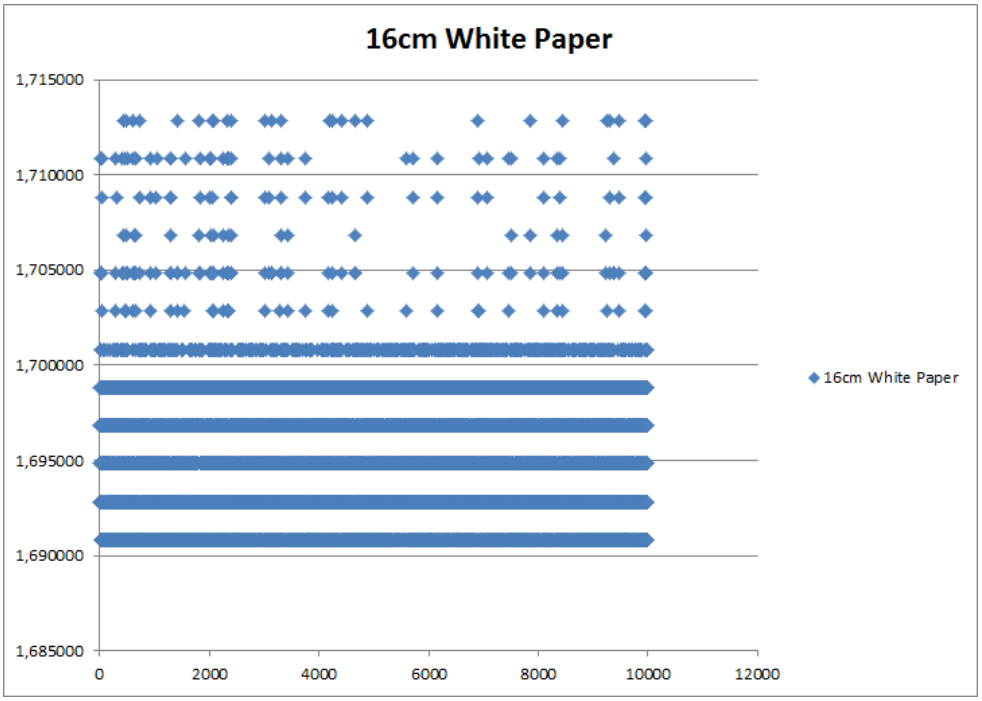


Figure 16.2: IR: 16 cm to white paper

Gathering 10 000 values while measuring the distance of 16cm on a white paper. We used the white paper to compare the values with them in the datasheet.

As we changed the distance and the surface, we discover a astonishing fact.

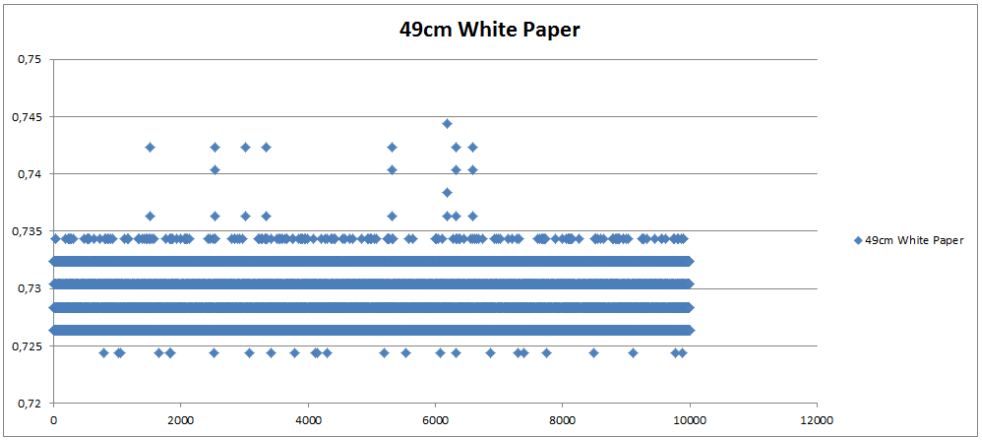


Figure 16.3: IR: 49 cm to white paper

Same measurement on the surface of the table:

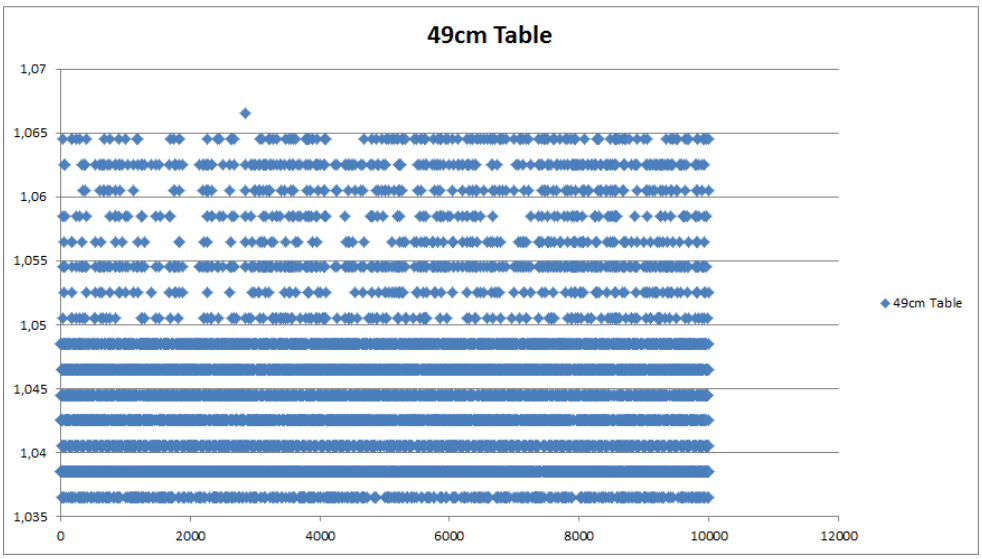


Figure 16.4: IR: 49 cm to table

As you see, the measurements on different surfaces are highly varying. At 49cm we got the following rough values:

White Paper: 0,73 V
Desk Surface: 1,42 V

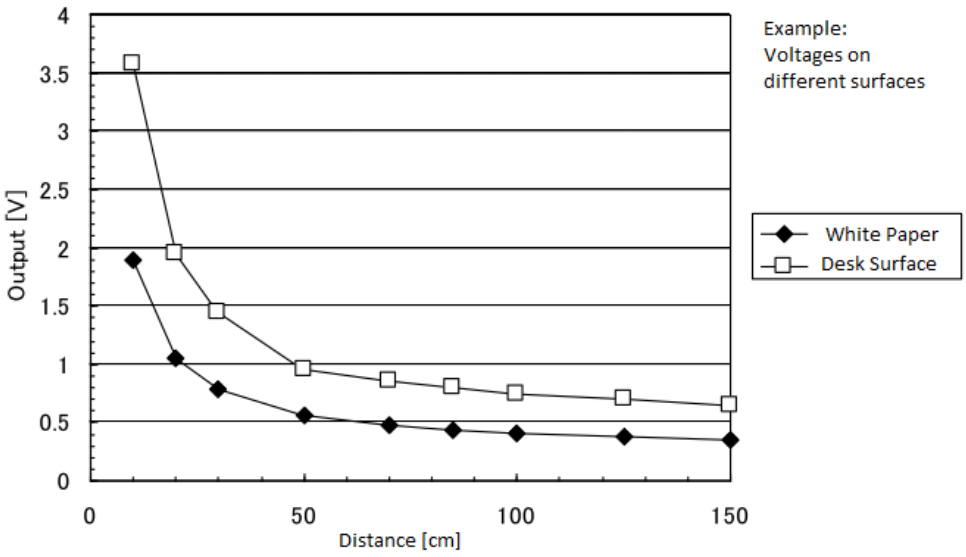


Figure 16.5: IR: Sensor values (voltages) on different surfaces

2 Surfaces and 2 Distances:

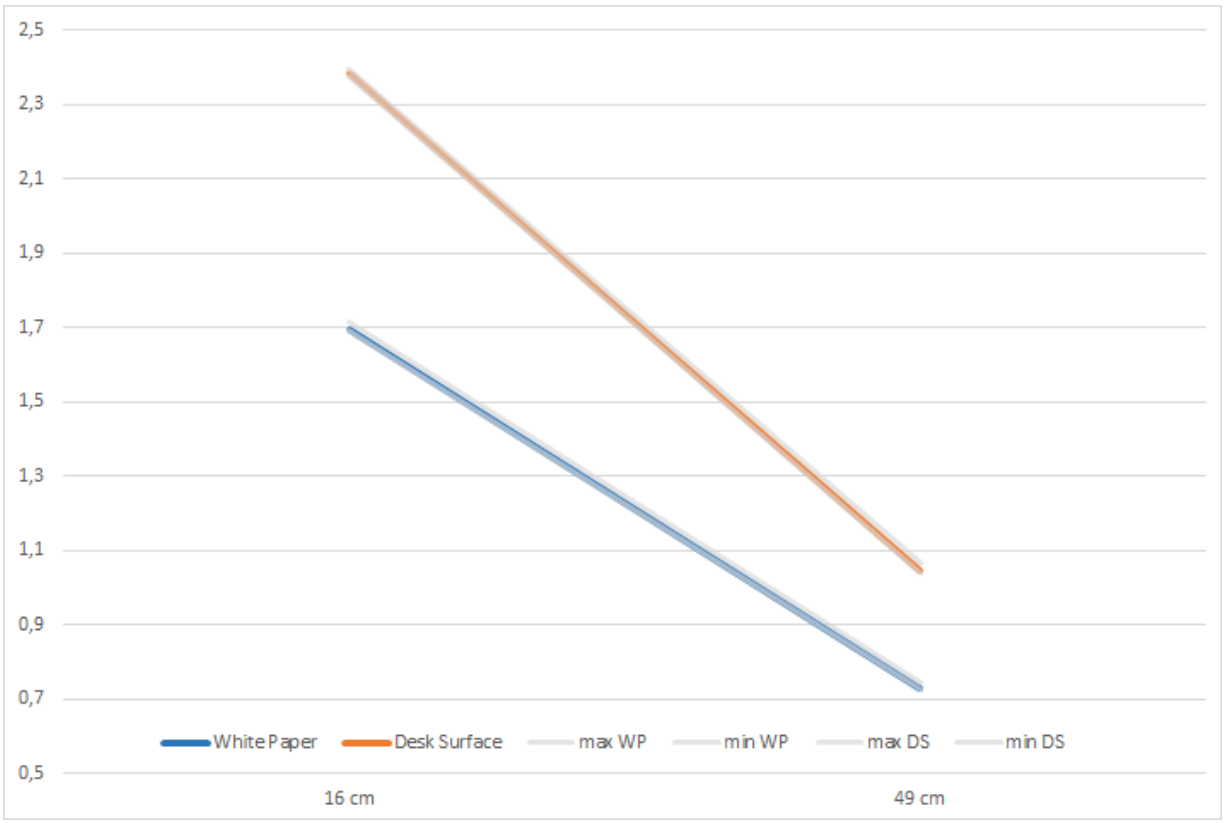


Figure 16.6: IR: Compared Voltages on two surfaces

We only measured 2 different distances in order to check the functionality of the sensor. The grey lines shows the jitter which we measured.

16.3 Conclusion

As we could not guarantee a good flying with changing surfaces, we decided to stop chasing a solution with the IR Sensor.

17 Ultrasonic Distance Sensor

Our next approach was, measuring the distance with a ultrasonic distance sensor. In a meeting we agreed on using 3 ultrasonic sensor to measure the distance to ground, because a single sensor is imprecise. We hoped that a sensor fusion of 3 ultrasonics will bring us better values.

While searching for theses sensors, we found a relatively cheap Laser Sensor. After a short consultation, we decided to go with that one. So Ultrasonic was discarded, before start.

18 LIDAR Laser Sensor

LIDAR-Lite Laser Distance Sensor
Model LL-905-PIN-01

Performance
Range: 0-20m LED Emitter
Range: 0-60m Laser Emitter

Interfaces
- I^2C
- PWM

For detailed technical values, please see the sensor chapter or the datasheet.

18.1 Connecting the sensor with I^2C

Because there is no proper input for a PWM signal, we used the I^2C Bus to connect the laser sensor. As the sensor does not support fast I^2C mode, which we need, we decided to use the second I^2C on the Raspberry Pi. This bus is orginally located on the camera port with an FPC (flexible printed circuit) Connector. We managed it to redirect it to the normal Pins on the board. See Chapter xxxxxx.

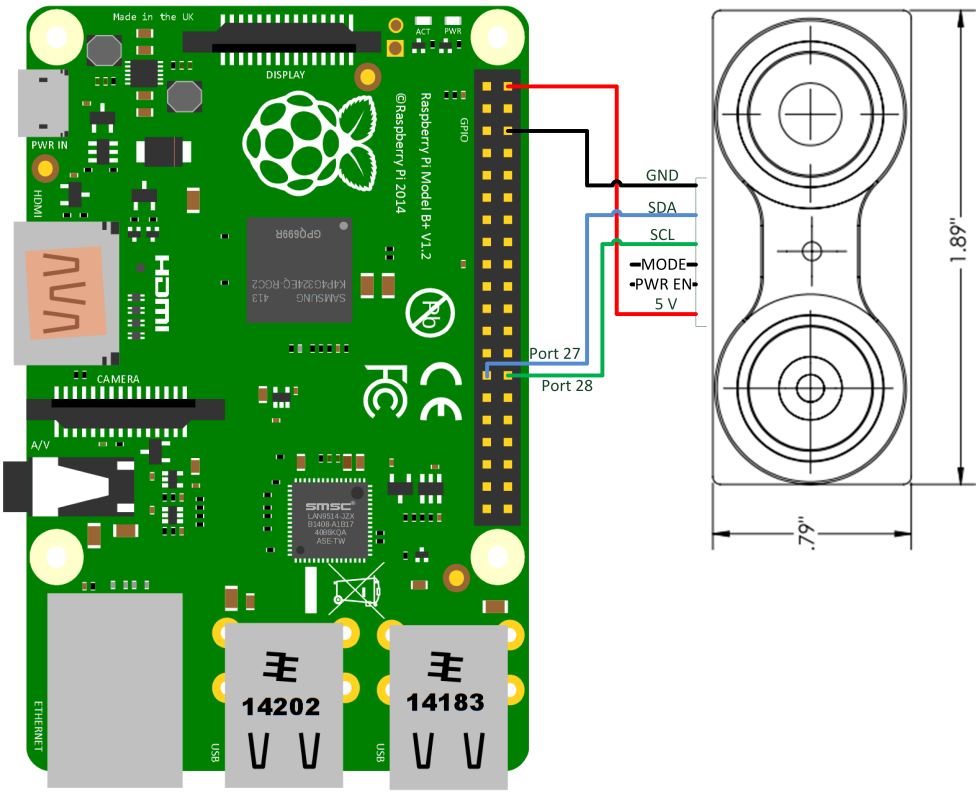


Figure 18.1: Wiring LIDAR

18.2 First steps

Quick Start Guide

1. Make Power and I2C Data Connections as per J1 connector pin out diagram. Pins 2 & 3 are optional connections and not required.
2. Initialization: Apply Power to the Module. The sensor operates at 4.75-5.5V DC Nominal, Maximum 6V DC.
3. Measurement: Write register 0x00 with value 0x04 (This performs a DC stabilization cycle, Signal Acquisition, Data processing). Refer to the section "'I2C Protocol Summary"' in this manual for more information about I2C Communications.
4. Periodically poll the unit and wait until an ACK is received. The unit responds to read or write requests with a NACK when the sensor is busy processing a command or performing a measurement. (Optionally, wait approx. 20 milliseconds after acquisition and then proceed to read of high and low bytes)
5. Read: register 0x0f, returns the upper 8 bits of distance in cm, register 0x10, returns the lower 8 bits of distance in cm. (Optionally a 2-Byte read starting at 0x8f can be done)

18.3 Measured Values

To test and verify functionality and accuracy of the sensor a series of tests was done. The following figure shows the deviation of the sensor of different surfaces. Each bar represents the mean value of at least ten measurements. As you can see at a distance of app. 150 cm the deviation is only 2 cm. With this tests the given accuracy of +/- 2.5 cm (see datasheet) gets confirmed.

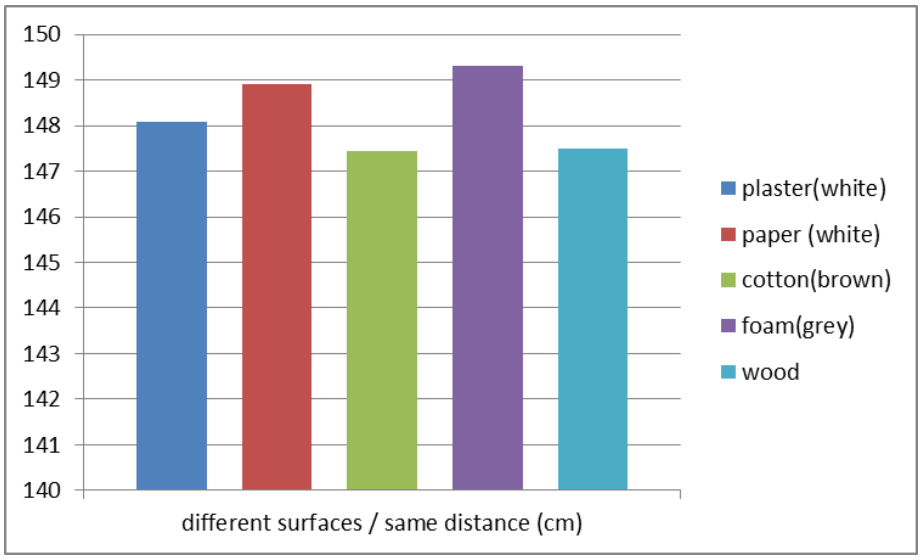


Figure 18.2: Laser measured values

To verify a correct distance measurement while flying also different angles to ground were tested. In a set of tests with an angle between 90°(vertically) and 10°and a distance range of app. 200cm the sensor returned valid and correct values.

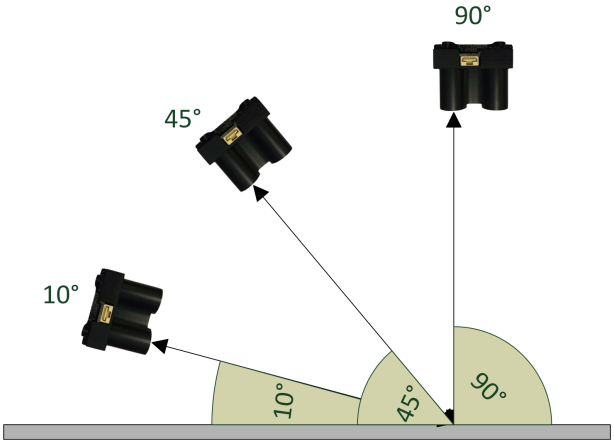


Figure 18.3: different measured angles

18.4 conclusion

The Laser sensor provides us reliable distance values, independent from the surface or material we measure at. The measurement get's a little bit worse below 20 cm, but with a estimated build in height of 10 cm at the quadrocopter, we can along with that.

We recommend to pursuit future autonomous landing approaches, with that sensor.

Regarding the Class 1 Laser Specification, there is no danger in using the sensor, even when looking straight in the laser.

19 Functions in C

These chapter is about the C-Code we developed to work via Raspberry Pi with the sensors.

19.1 I²C

```
1 unsigned int g_lldI2c_WriteI2c_bl(unsigned char, const unsigned
    char*, unsigned int);
```

Listing 19.1: Write on I2C-1

Parameter:	unsigned char	slave address of device
	const unsigned char*	buffer with data to write
	unsigned int	number of bytes to write
Return:	unsigned int	error detection, 0=OK 1=failure
Description:	function to write several data on the I2C-1 Bus of the Raspberry PI	

```
1 unsigned int g_lldI2c_ReadI2c_bl(unsigned char, const unsigned char
    *, unsigned int);
```

Listing 19.2: Read from I2C-1

Parameter:	unsigned char	slave address of device
	const unsigned char*	array to store read data
	unsigned int	number of bytes to read
Return:	unsigned int	error detection, 0=OK 1=failure
Description:	function to read several data on the I2C-1 Bus of the Raspberry PI	

```
1 unsigned int g_lldI2c_WriteI2c0_bl(unsigned char, const unsigned
    char*, unsigned int);
```

Listing 19.3: Write on I2C-0

Parameter:	unsigned char	slave address of device
	const unsigned char*	buffer with data to write
	unsigned int	number of bytes to write
Return:	unsigned int	error detection, 0=OK 1=failure
Description:	function to write several data on the I2C-0 Bus of the Raspberry PI	

```
1 unsigned int g_lldI2c_ReadI2c0_bl(unsigned char, const unsigned
    char*, unsigned int);
```

Listing 19.4: Read from I2C-0

Parameter:	unsigned char	slave address of device
	const unsigned char*	array to store read data
	unsigned int	number of bytes to read
Return:	unsigned int	error detection, 0=OK 1=failure
Description:	function to read several data on the I2C-0 Bus of the Raspberry PI	

19.1.1 Configuration

$\#include < linux/i2c-dev.h >$ is doing the (local) device handling. We only use the slave address to communicate with the connected devices. Read or write commands are provided by fcntl.h.

19.1.2 I²C Write

Writes the number of stated bytes from the write buffer with the "write" command to the chosen I²c-device, depending on called function.

19.1.3 I²C Read

Reads the number of stated bytes from the read buffer with the "read" command from the chosen I²c-device, depending on called function.

19.2 Analog-Digital-Converter (ADC)

```
1 float g_halADC_get_ui16(unsigned char );
```

Listing 19.5: Read from ADC

Parameter:	unsigned char	A0-A3 input selection
Return:	float	converted analog values
Description:	Interface to read ADS1015 (ADC)	

19.2.1 Configuration

Sensor Board Name: Pololu ADS1015
Sensor Name: GP2Y0A60SZLF

l_mux_ui8 = 0xC2;
" C "$_{16}$:
The first Hex-Value depends on Starting Conversion + the Input, which Pin to read A0-3

" 2 "$_{16}$:
The second Value is PGA (001)=+-4,099V and continuous Mode (0)

These three bytes are written to the ADS1015 to set the config register and start the conversion

l_writeBuf_rg24[0] = 1;
This sets the pointer register to write the following two bytes to the config register

l_writeBuf_rg24[1] = l_mux_ui8;
This sets the 8 MSBs of the config register (bits 15-8) to 11000011

l_writeBuf_rg24[2] = 0x23;
This sets the 8 LSBs of the config register (bits 7-0) to 00100011

" 2 "$_{16}$:
// First Hex is sample Rate. (001) sets to 250SPS + Comp Mode (0)

" 3 "$_{16}$:
// Second Hex is Comp. config. (0011) disable the comparator

The l_writeBuf_rg24 is written to the configuration register of the ADC. After that, we can read the converted analog values from the chosen input.

19.2.2 ADC Read

To read a converted analog value, you need to set the pointer register to 0.

When the pointer register is set to 0 this signals that the converted analog value should be provided. You will get the these values when performing a i^2c-read command the next time. This value is 16 Bit large and will be calculated as a float-value, depending on the resolution which is adjusted.

19.3 Infrared Sensor

```
1 float g_halADC_get_ui16(unsigned char );
```

Listing 19.6: Read Infrared

Parameter: char select Input on which IR is connected

Return: float Voltage from Sensor

Description: Since our IR Sensor only provides analog output, we need to use the ADC

19.3.1 Configuration

There is no configuration of the Infrared sensor needed.

19.3.2 Read Sensor Values

We get the analog values with the ADC-Function.

19.4 LIDAR-Lite Laser Sensor

```
1 double g_LIDAR_getDistance_f64(void);
```

Listing 19.7: get Laser Distance

Parameter: void

Return: double distance in meter

Description: returns calculated distance value in meters

```
1 int g_LIDAR_readDistanceFromI2C_i32(void);
```

Listing 19.8: trigger Laser measurement

Parameter: void

Return: int error detection, 0=OK -1=failure

Description: triggers a measurement and stores the result

19.4.1 Configuration

Trigger Measurement of Distance (DC stabilization cycle, Signal Acquisition, DataProcessing)

First Config Byte: 0x00;
Representation of configuration register 0x00 of the laser sensor.
Second Config Byte: 0x04;
Take acquisition and correlation processing with DC correction

Set Reg 0x8f as Output-Register to read a two-byte value which gives the distance in cm.

19.4.2 Read Sensor Values

Read two-byte distance in cm from register 0x8f. This value is stored as a decimal value in cm.
Alternative you can read the high-byte of the measured value from register 0x0f and the low-byte from register 0x10.

20 I²C Configuration

This chapter describes all necessary steps to get the two I²C buses of the Raspberry Pi (Model B+) up and running. Because of different operating modes of the devices using the I²C-Bus the usage of both buses is necessary.

HINT: These steps are not necessary if you install the Rasbian Image of the project. Within that image, everything should be configured.

20.1 raspi-config

Enable I²C using raspi-config utility.

From the command line type:

```
sudo raspi-config
```

This will open the raspi-config utility.

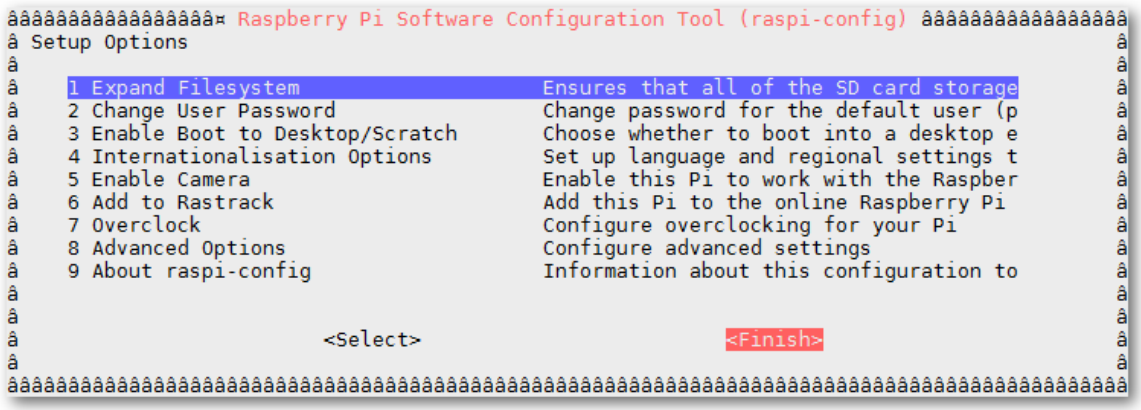


Figure 20.1: raspi-config

Now complete the following steps:

Select: "8 - Advanced Options"
Select: "A7 - I²C"
Select: "Yes"
The Screen will ask if you want the interface to be enabled:
Select: "Yes"
Select: "OK"
The Screen will ask if you want the module to be loaded by default:
Select: "Yes"
The Screen will state the module will be loaded by default:
Select: "OK"
Select "Finish" to return to the command line
When you next reboot the I²C module will be loaded.

20.2 Module File

Next we need to edit manually the modules file using:
`sudo nano /etc/modules`
and add the following lines:
`i2c-bcm2708`
`i2c-dev`
Use CTRL-X, then Y, then RETURN to save the file and exit.

20.3 I²CTools

For hardware monitoring, device identification, and troubleshooting we install "i2c-tools".

```
sudo apt-get update
sudo apt-get install i2c-tools
```

Now shutdown your system, disconnect the power to your Pi and you are ready to connect your I²C-hardware.

20.4 Test I²C-1

Check if I²C is enabled:

When you power up or reboot your Pi you can check the I²C module is running by using the following command:
`lsmod | grep i2c_`
That will list all the modules starting with "i2c_". If it lists "i2c_bcm2708" then the module is running correctly.

Testing Hardware:
Once you've connected your hardware double check the wiring. Make sure 5V is going to the correct pins and you've got not short circuits. Power up the Pi and wait for it to boot. Then type the following command:

```
sudo i2cdetect -y 1
```

With e.g. a sensor connected the output looks e.g. like this:

```
    0 1 2 3 4 5 6 7 8 9 a b c d e f
00:         - - - - - - - - - - - - -
10:  - - - - - - - - - - - - - - - -
20:  - - - - - - - - - - - - - - - -
30:  - - - - - - - - - - - - - - - -
40:  - - - - - - - - - - - - - - - -
50:  - - - - - - - - - - - - - - - -
60:  - - 62 - - - - - - - - - - - -
70:  - - - - - - - -
```

This shows that one device is connected and its address is 0x62.

20.5 Set up I²C-0

In normal configuration the second I²C-Bus of the Raspberry Pi is set up as two of the output pins of the DSI Display Connector resp. the CSI Camera Connector.
To make the setup of the quadrocopter as easy as possible and with respect to the weight and soldering/cabling these output pins were redirecteed to two of the 40 pins of the GPIO Header.

This gets done by useage of a Python-script (see below) which gets excecuted while booting the system. To get this configuration running two additional files need to be edited.

In "/boot/cmdline.txt"
`bcm2708.vc_i2c_override=1`
has to be added
in "/etc/modprobe.d/i2c_o_enable.conf"
`blacklist snd_soc_tas5713`
has to be added.

After this the GPIO port 27 is configured as SDA0 and the GPIO port 28 as SCL0.

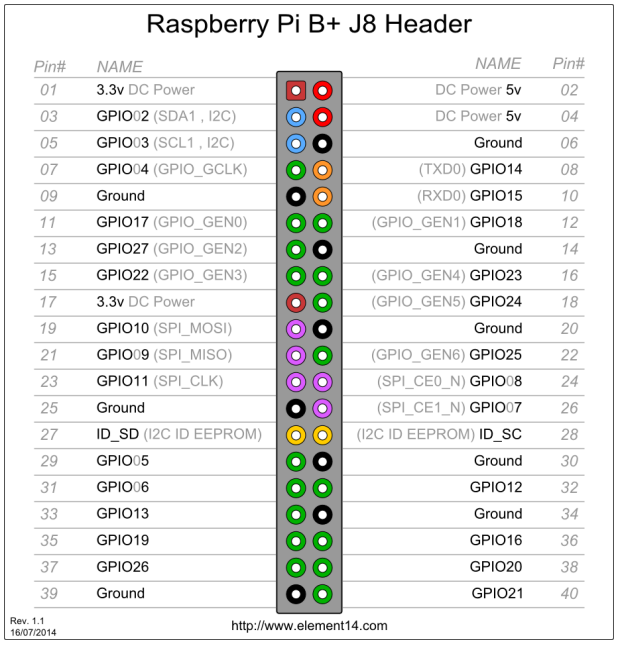


Figure 20.2: GPIOs [2]

2 http://www.element14.com/community/servlet/JiveServlet/previewBody/68203-102-6-294412/GPIO.png

```
#!/usr/bin/python
#!/usr/bin/env python
#
#

# #######
# For I2C configuration test
import os
import mmap
bplus=0
BCM2708_PERI_BASE=0x20000000
GPIO_BASE=(BCM2708_PERI_BASE + 0x00200000)
BLOCK_SIZE=4096

def _strto32bit_(str):
    return ((ord(str[3])<<24) + (ord(str[2])<<16) + (ord(str[1])<<8) + ord(str[0]))

def _32bittostr_(val):
    return chr(val&0xff) + chr((val>>8)&0xff) + chr((val>>16)&0xff) + chr((val>>24)&0xff)

def get_revision():
    with open('/proc/cpuinfo') as lines:
        for line in lines:
            if line.startswith('Revision'):
                return int(line.strip()[-4:],16)
    raise RuntimeError('No revision found.')

def i2cConfig():
    if get_revision() >= 10:
       print "B+ or CM detected."
       s0 = 0b00000000000000000000000100100100
       s2 = 0b00000000000000000000000000000000
    if get_revision() <=9:
       s0 = 0b00000000000000000000000100100000
       s2 = 0b00100100000000000000000000000000
    if get_revision() <= 3:
        print "Rev 2 or greater Raspberry Pi required."
        return
    # Use /dev/mem to gain access to peripheral registers
    mf=os.open("/dev/mem", os.O_RDWR|os.O_SYNC)
    m = mmap.mmap(mf,BLOCK_SIZE, mmap.MAP_SHARED,
            mmap.PROT_READ|mmap.PROT_WRITE, offset=GPIO_BASE)
    # can close the file after we have mmap
    os.close(mf)
    # Read function select registers
    # GPFSEL0 -- GPIO 0,1 I2C0   GPIO 2,3 I2C1
    m.seek(0)
    reg0=_strto32bit_(m.read(4))
    # GPFSEL2 -- GPIO 28,29 I2C0
    m.seek(8)
    reg2=_strto32bit_(m.read(4))
    # print bin(reg0)[2:].zfill(32)[2:]
    # print bin(reg2)[2:].zfill(32)[2:]

    # GPFSEL0 bits --> x[26] SCL0[3] SDA0[3]
    #                          GPIO    GPIO
    m0 = 0b00000000000000000000111111111111
    #s0 = 0b00000000000000000000100100100100
    b0 = reg0 & m0
    if b0 <> s0:
        #print "reg0 I2C configuration not correct. Updating."
        reg0 = (reg0 & ~m0) | s0
        m.seek(0)
        m.write(_32bittostr_(reg0))

    # GPFSEL2 bits --> x[2] SCL0[3] SDA0[3] x[24]
    m2 = 0b00111111100000000000000000000000
    b2 = reg2 & m2
    if b2 <> s2:
        #print "reg2 I2C configuration not correct. Updating."
        reg2 = (reg2 & ~m2) | s2
        m.seek(8)
        m.write(_32bittostr_(reg2))

    # No longer need the mmap
    m.close()


if __name__ == '__main__':
    i2cConfig()
```

Listing 20.1: I²C0 Port-Configuration

21 Conclusion and outlook

21.1 Achieved project goals and results

First of all a project plan for the two groups was set up and the needed hardware was chosen. Following this a plan of the mounting and wiring was drawn and two prototypes were build up.
To increase the flexibility of using various operating systems on the development computer an Ubuntu system on a virtual machine is used. Programming is done by Eclipse which also is installed in the virtual machine. Additionally cross compiling and the ability to debug is enabled. Due to the fact that the compiling runs on the development computer, the speed of compiling is dramatically increased. The remote debugging feature helps to track down and fix errors during the development process. The already set up development environment can be downloaded from the SVN repository.
To ensure real-time capability, a real time patch is applied to the Raspbian distribution of the Raspberry Pi.
By introducing of hierarchies in the software, the hardware dependency is separated to just one layer. This makes the software better portable. In case of hardware changes, only in the affected layer software has to be changed. With this in mind, it was split up in Low level driver, the hardware abstraction, signal processing and application layer.
Beginning with the programming an interface abstraction to the Low level drivers of the UART was developed and proofed. The also needed interface abstraction of the I2C driver was implemented by the second project group. Just some improvements were done by the authors of this document. The Graupner PPM decoding is enabled via a Kernel module which can be loaded directly to the Kernel of the system. This enables interrupting with a very low jitter. This is needed to provide accurate measuring of the PPM pulses of the remote control. Additionally an experimental time trigger is provided via a second Kernel module driver. This is used to provide accurate timing for the control loop. The patched and pre-configured operating system of the Raspberry Pi can also be downloaded from the SVN repository.
After successful testing of the low level drivers in conjunction with the interface abstraction, the development of the hardware abstraction layer was started. The implementation of the GPS driver and the Inertial measurement unit sensors was started and successfully finished. All of these modules provide an data interface for the next hierarchy. All the drivers are tested as single units to ensure proper working.
The last step of the implementation was the signal processing layer. On this hierarchy an reduced IMU-Filter and the Orientation fusion was implemented.

The sensor fusion covers the successful implementation of the complementary Filter and the Kalman Filter. Additionally a generic matrix library was programmed to enable Matrix operations for the Kalman Filter. This library can be used with matrices of various sizes, written in pure C-code. The sensor fusion provides the absolute orientation of the system. The orientation angles of roll, pitch and yaw are with the successful fusion independent of the restrictions of the sensors and provide accurate angles.
A Matlab model was produced which gets data from the Raspberry Pi via UDP network connection. Just a network cable needs to be connected between the host computer and the Raspberry Pi. On the Raspberry Pi is a C library used which enables the communication. With the help of this part the implementation can be tested and the configuration of the filter parameters can be improved.
Finally a doxygen file was written for automatic code documentation.

21.2 Remaining project goals and outlook

The basis of the PPM measurement is provided via a Kernel module. The analysis of the measured time differences needs to be done to get the separated control signals.
The orientation fusion delivers perfect representation of all three angles around the X,Y and Z axis of the system. Because of the usage of Euler angles there can be a gimbal lock when due to rotations two of the three axis fall together. Then one degree of freedom is lost. To solve this problem the system should not use Euler angles, instead Quaternion needs to be used.
Autonomous flying with just the orientation is not possible. Also the position of the Quadrocopter needs to be known. For this, additional sensor fusion focusing the velocity and position in all directions has to be calculated.
Last step is the implementation of the controller for the orientation and the position which finally controls the complete system. To switch between flying with the remote control and the autonomous flying a switch of the remote control needs to be used.

Bibliography

[BCM] BCM2835 ARM Peripherals; Broadcom Corporation, 2012 https://www.raspberrypi.org/documentation/hardware/raspberrypi/bcm2835/BCM2835-ARM-Peripherals.pdf

[BorEng] Alex, The quadcopter : control the orientation, 2012, http://theboredengineers.com/2012/05/the-quadcopter-basics/

[LMK69] Jürgen Quade, Eva-Katharina Kunst; Kern-Technik: Kernel- und Treiberprogrammierung mit dem Linkux-Kernel (Folge 69); Linux Magazin 08/2013 http://www.linux-magazin.de/Ausgaben/2013/08/Kern-Technik

[LMK70] Jürgen Quade, Eva-Katharina Kunst; Kern-Technik: Kernel- und Treiberprogrammierung mit dem Linkux-Kernel (Folge 70); Linux Magazin 10/2013 http://www.linux-magazin.de/Ausgaben/2013/10/Kern-Technik

[LMK81] Jürgen Quade, Eva-Katharina Kunst; Kern-Technik: Kernel- und Treiberprogrammierung mit dem Linkux-Kernel (Folge 81); Linux Magazin 07/2015 http://www.linux-magazin.de/Ausgaben/2015/07/Kern-Technik

[REP] Robotics/Electronics/Physical Computing; Wordpress Blog, https://trandi.wordpress.com/2011/04/12/graupner-r700-ppm-signal/

[RPL] Robot Platform; Wordpress Blog http://www.robotplatform.com/knowledge/servo/servo_control_tutorial.html

[STM] Jay Esfandyari, Roberto De Nuccio, Gang Xu; STMicroelectronics, http://uk.mouser.com/applications/sensor_solutions_mems/