Quadrocopter stabilization using neuromorphic Embedded Dynamic Vision Sensors (eDVS)

eingereichte
FORSCHUNGSPRAXIS
von

B.Sc. Florian Bergner

geb. am 30.09.1988
wohnhaft in:
Linprunstr. 06
80335 München
Tel.: 089 78794156

Lehrstuhl für
STEUERUNGS- und REGELUNGSTECHNIK
Technische Universität München

Univ.-Prof. Dr.-Ing./Univ. Tokio Martin Buss
Univ.-Prof. Dr.-Ing. Sandra Hirche

Betreuer:	M.Sc. Cristian Axenie
Beginn:	27.08.2012
Abgabe:	14.11.2012

Contents

Chapter 1

Introduction

1.1 Motivation

Quadrocopter architectures have been developed as efficient implementations of autonomous aerial vehicles. For these vehicles, system stability is a major concern. A previous practical project [1] introduced a new approach for stabilization, in which an event based vision sensor (eDVS) is used to provide additional motion information. Such eDVS sensors allow an efficient calculation of optic flow, which can be used to measure displacements in the pitch and roll axes of the project's quadrocopter (AR.Drone). A drawback of the current approach [1] is that the eDVS sensors attached to the quadrocopter do not directly communicate with the AR.Drone control system; but instead use a WLAN WiFi-module (IEEE 802.11g) to communicate via PC with the AR.Drone. Such a setting suffers from inevitable problems such as high latency or even data loss in the wireless connections.

1.2 Task

The goal of this Forschungspraxis is to design and implement a completely autonomous system, where on-board eDVS sensors connect directly to the AR.drone's controller. Consequently, the optic flow and control algorithms must be implemented on an embedded system on the AR.Drone. The AR.Drone only connects to a PC for setup or monitoring purposes. Two different on-board approaches are envisioned: (A) an embedded Linux installed on the AR.Drone can be adjusted to compute optic flow information from raw sensor data to modulate the existing on-board control decisions. Or (B) an independent embedded processing system can get added to the AR.drone to receive raw sensor data, compute optic flow and provide abstract correction signals for the existing motor controller. Both approaches shall get evaluated in theory and with a small test implementation. The more appropriate system shall get implemented to evaluate the system's performance with and without additional stabilization due to optic flow.

1.3 Chronological sequence

week 1 setup and configuration of AR.Drone, eDVS sensor, and review of project [1], develop interface software (TCP) to communicate with AR.drone

week 2 investigate programming of on-board Linux / evaluate alternative embedded processing hardware, make decision to use approach (A) but design own controller rather than using the somewhat limiting OEM controller

week 3 write drivers for hardware components on AR.Drone, get raw sensor data, send commands to motors

week 4 process raw sensor data, implement Kalman Filter for sensor fusion to get reasonable roll and pitch angles, find relation between raw ultrasonic height sensor output and actual hight in cm

week 5 design attitude controller to stabilize roll, pitch, yaw and height of drone, find parameters for 4 PID controllers, examine controller performance

week 6 design dual port UART to USB converter to connect eDVS sensors directly to embedded system, cross-compile USB-OTG driver for embedded linux system, write eDVS driver for embedded linux system

week 7 try existing optic flow algorithm on embedded system, existing optic flow algorithm is far to slow, optimize speed of optic flow algorithm

week 8 write small matlab program to verify optimized optic flow algorithm, try to capture events and the optic flow while AR.Drone is flying, detect serious problem: attitude controller and optic flow algorithm work alone but not in combination

week 9 detect source of problem: motors are disturbed when USB-OTG driver module is loaded into the linux kernel, loading the module causes a collapse in the 5V supply line, motors hang up and can only be reinitialized by rebooting the embedded linux system

1.4 Approach

Before I explain what approach option (A or B) I've finally selected I want to show my investigation results about the hardware and software of the AR.Drone.

1.4.1 Investigation results

After investigating the hardware and software of the AR.Dorne it became rapidly clear that the stabilizing controller which is delivered with the AR.Drone is somewhat limited. That controller only accepts commands like "Move forward/backward", "Move to the left/right" and "Move up/down". Also the controller provides attitude information only with a quite slow update rate (about 20 ms). Sometimes the controller even shows unexplainable behaviors.
But to get good controlling results it is desirable to get a tighter grip on the drone. For example it would be useful to get attitude data with a higher update rate and to set attitudes directly by appling set values for roll, pitch and yaw angles to the controller.
Tests also show that the embedded linux of the AR.Dorne already has a big workload to cope with. Besides managing hardware resources the embedded linux on the drone runs one big program which processes sensor information, camera image inputs, user inputs and the main stabilizing controller. Unfortunatelly that program is closed source and consequently cannot be modified. Even worse: The remaining computing power on the drone is not sufficient; not even for a resource preserving optic flow algorithm in combination with eDVS sensors.
So when using the OEM stabilizing controller program on the drone the project can only be realized by adding an external embedded processing unit onto the drone. This implies additional weight and costs.
Further investigations on hardware and embedded linux show that it is rather easy to access the drone's sensors and motors directly. How to access motors and sensors is shown by tutorials in the internet [2]. So it is definitively feasable to construct our own attitude controller. This controller stabilizes only roll, pitch, yaw and height and neglects camera inputs of the drone completely. In this way more than enough computing power should be available for processing optic flow in combination with eDVS sensors.

1.4.2 Approach selection and realization steps

The investigation results show that approach option A in combination with a self developped attitude controller is the better choice. This way no external embedded processing unit is needed and the optic flow calculations can be done directly on the embedded linux of the drone. The drone has also an USB OTG port which can be used to connect the eDVS sensors.
As it can be seen in figure 1.1 the final stabilization system consits of three main

parts:

- Attitude Controller
- eDVS Driver and Optic Flow Algorithm
- Horizontal Drift Compensation Controller

So it makes sense to structure the approach into three tasks. These tasks are dealt with in the following three chapters.

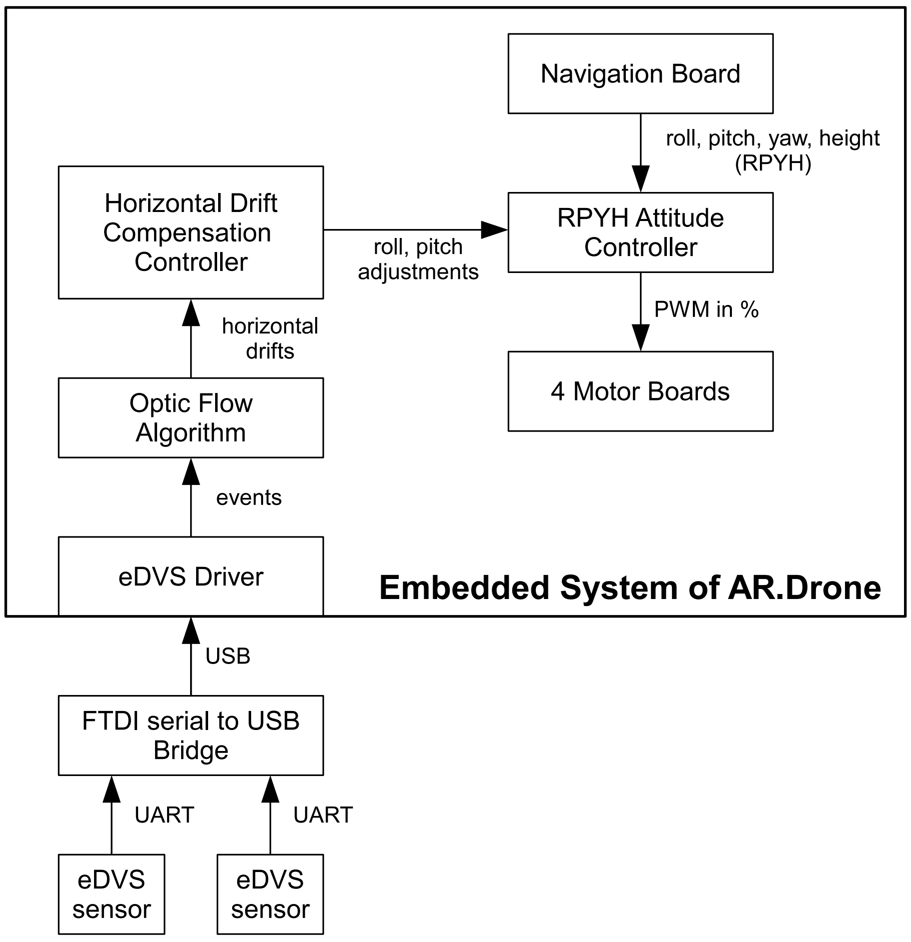

Figure 1.1: System Overview

Chapter 2

Attitude Controller Design for AR.Drone

In this chapter I show how I implement my own attitude controller for the AR.Drone. This attitude controller tries to stabilize the roll, pitch and yaw angle and the height of the drone. The controller uses the gyroscope, the accelerometer and the ultrasonic altimeter of the ardrone as inputs. The outputs of the controller are PWM commands for the four drone motors. See illustration in figure 2.1.
Before we have a closer look to the details of the attitude controller I want to summarize briefly the hardware and software properties of the drone.

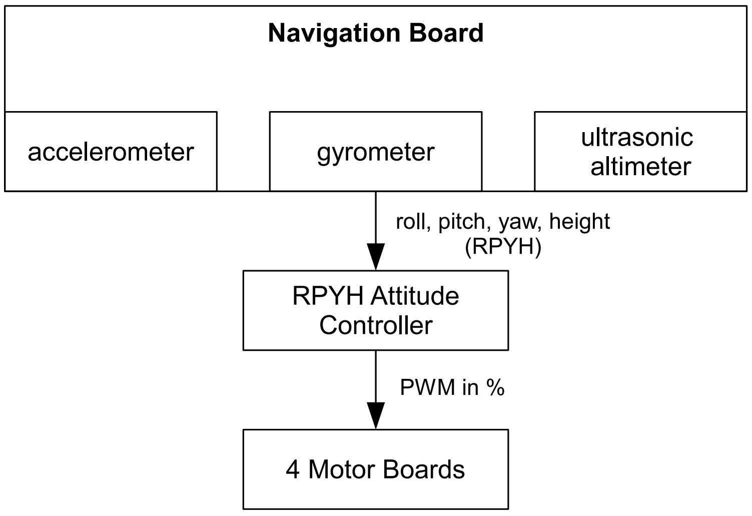

Figure 2.1: Attitude Controller Overview

2.1 AR.Drone Embedded System

2.1.1 Hardware Components

The information about the hardware components of the AR.Drone is taken from the sources [3] and [2].
The AR.Drone consists of 3 clusters of hardware components.

- 4 motor boards
- 1 motherboard
- 1 navigation board

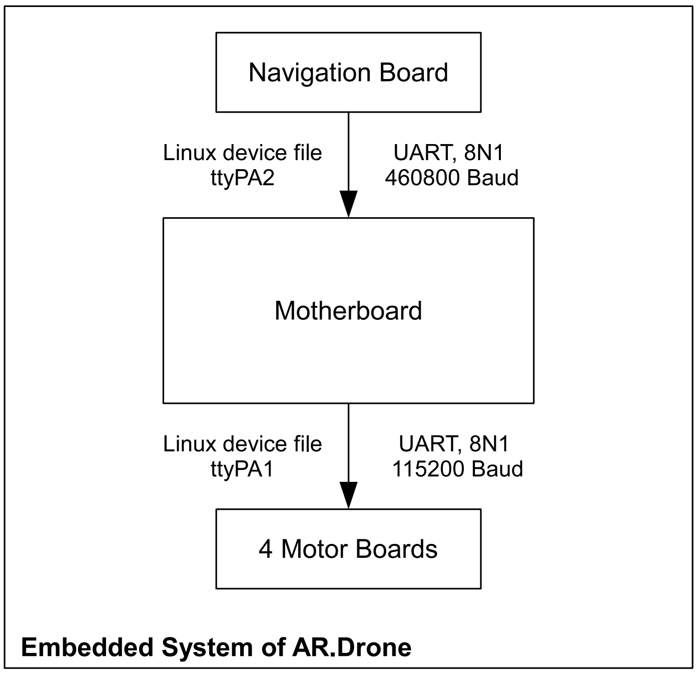

Figure 2.2: Embedded System Overview

4 motor boards Each of the four motor boards is located directly underneath the propellers, has a burshless motor for thrust generation and it's own motor controller. The motor controller is realized by using ATmega8A 8bit Microcontrollers. The motor controlling system has implemented a cutout system which prevents damages when obstacles get into the motors.
Each motor board has its own interrupt line to signal such cutout events to the

mainboard. For communication purposes the 4 motor boards share one bidirectional UART connection to the main board. The UART connection uses a baudrate of 115200 Baud with a 8N1 data format. The UART to the motorboards can be interfaced by using the linux device file "ttyPA1". See figure 2.2 to get a better overview of the connections between the motherboard and the motor boards.
After the initialization process the motors await an update of their PWM values every 5 ms (200 Hz).

Motherboard The motherboard is the main board of the AR.Drone. The navigation board and the 4 motor boards are connected to it. The motherboard supplies these boards with the needed voltages, captures sensor information from the navigation board and sends commands to the motor boards.
On the mainboard itsself the following hardware blocks are implemented.

- Parrot P6 32bit ARM9 Core (468 Mhz, 128 MB RAM)
- WiFi connection
- 2 cameras connected to the motherboard:
  - bottom camera (60 FPS)
  - front camera (15 FPS)
- USB OTG connection

The Parrot P6 processor runs an embedded realtime linux system which processes all the incomming data. What kind of things the system actually processes will be explained in section 2.1.2.

Navigation board The main task of the navigation board is to read out data from the sensors, convert the data to digital raw values and finally send the raw data via UART to the mainboard. The navigation board consists of the following components.

- 16bit PIC Controller 40 Mhz
- 3 axis accelerometer
- 2 axis gyrometer and a 1 axis gyrometer
- ultrasonic altimeter

The accelerometer and the gyrometers have an update rate of 200 Hz and the ultrasonic altimeter has a range of 6 m and an update rate of 25 Hz. The UART connection uses a baudrate of 460800 and a 1N8 data format. The linux device file of that UART is "ttyPA2".

2.1.2 Realtime Embedded Linux

The embedded linux runs several threads simultanously. These threads are:

- managing WiFi connections
- video data sampling
- video data compression for WiFi transmission
- image processing
- sensor data acquisition
- attitude estimation
- control loop at 200 Hz

Besides managing WiFi connection all these threads are controlled by one single main program. This main program has the meaningful name "program.elf". The main program is started automatically with every reboot of the system.
The terminal of the embedded linux can be accessed by connecting to the wireless LAN network of the drone and running "telnet 192.168.1.1" on the host PC. The file system of the drone can be accessed by using a ftp connection to the same ip-address. System information logs are sent to a seperate UART connection. This connection can also be accessed by the linux device file "ttyPA0".

Vision algorithm The vision algorithm is performed on images from the bottom camera. The goal of this algorithm is to estimate the horizontal velocity of the drone with respect to the ground. This velocity cannot be estimated by just using the accelerometer and the gyroscope.
The vision algorithm performs two approaches. It tries to estimate the horizontal velocity by computing the optic flow on images from the bottom camera. At the same time it applies corner detection on these images. The algorithm tracks these corners and calculates their displacement over time.
As you can imagine the vision algorithm needs a lot of computing power.

2.1.3 OEM Free Flight Controller

The main drone controlling program is called Free Flight Controller. This program runs in the "program.elf" on the embedded linux system of the Drone. The Free Flight Controller listens for controlling commands (AT-commands) on the wireless UDP port 5556 and sends processed sensor data and attitude estimations through the wireless UDP port 5554. The AT-commands must be repeated in an intervall of at most 50 ms. Otherwise the connection watchdog detects a lost connection and the drone performs an emergency landing. The sensor information is sent with an update rate of about 50 Hz. Nevertheless the UDP protocol has no handshake rules and packets are sometimes lost. The wireless connection causes also some latency.

2.2 Custom Attitude Controller Design

The attitude controller uses sensor data to stabilize the drone in 4 degrees of freedom (DOFs) (See figure 2.1 and 2.3). The DOFs are roll, pitch, yaw angles and the height. Unfortunately the raw sensor data must be preprocessed before it can be used by the attitude controller. The raw sensor data which is represented by ADC values must be converted into angles in radians and distance in meters. Also for each sensor the zero point must be demermined. As the zero point for angles and height is depended on the drone setup the sensor data must be trimmed on every startup sequence of the attitude controller.

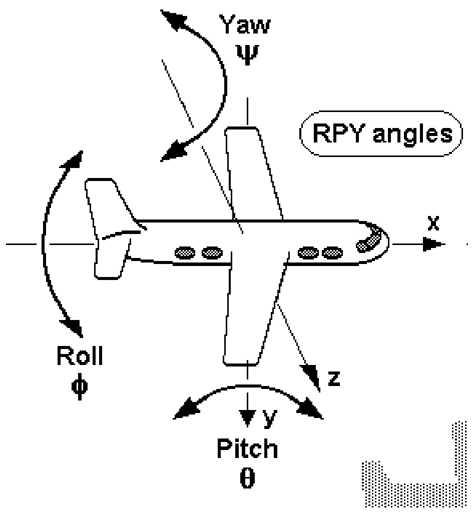

Figure 2.3: Roll Pitch Yaw (RPY) [4]

2.2.1 Acquiring roll, pitch and yaw angles (RPY)

The roll and pitch angles can be acquired in two ways while the yaw angle can only dermined in one way.

Acquiring RPY using the gyroscope The gyroscope measures turning rates in rad/s. When the turning rates are integrated one get the absolute RPY angles

ϕ, θ, ψ in rad.

$$\phi = \int \dot{\phi}\,\mathrm{d}t \tag{2.1}$$

$$\theta = \int \dot{\theta}\,\mathrm{d}t \tag{2.2}$$

$$\psi = \int \dot{\psi}\,\mathrm{d}t \tag{2.3}$$

Nevertheless this approach has one big disadvantage. We integrate here over noisy data so that after some time we accumulate errors and the calculated angle drifts away from the actual attitude angle of the drone (See figure 2.4).
The advantage in using the gyroscope is that it reacts very fast to changes. Also the yaw angle can only be dermined by using the gyroscope. It cannot be determined by using the accelerometer.

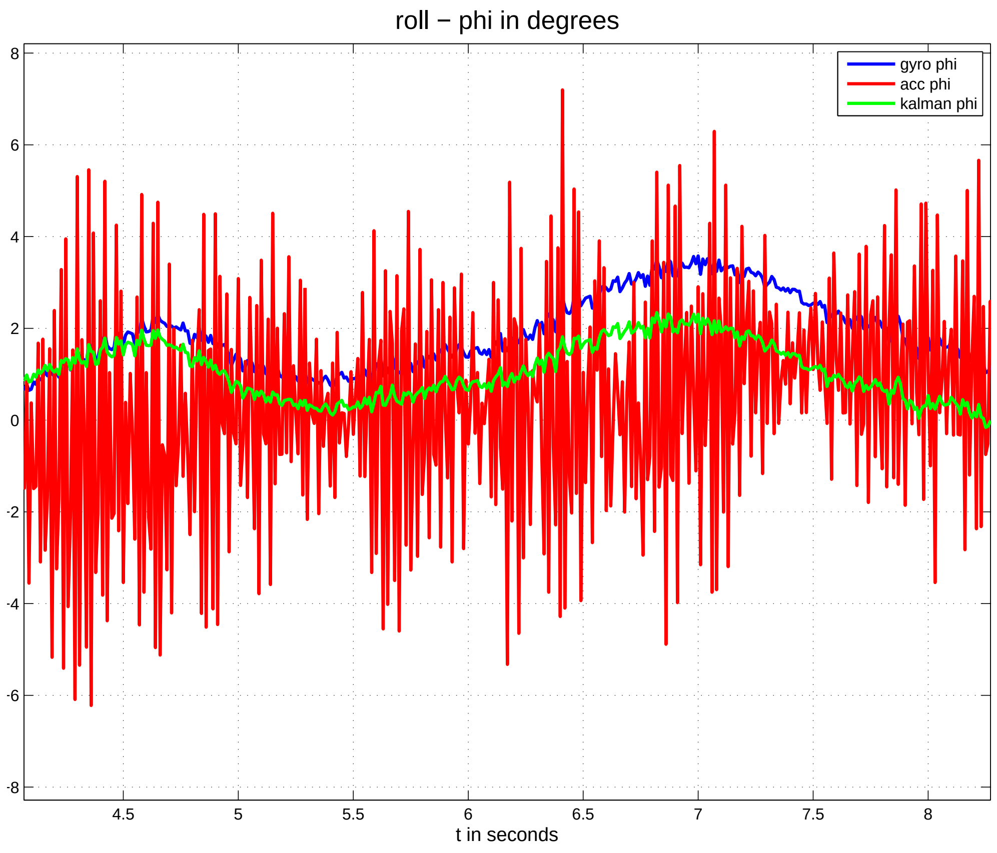

Figure 2.4: Roll Angle Measurement

Acquiring RP using the accelerometer The accelerometer measures acceleration into the X, Y, Z directions in g. Knowing that the graviational acceleration always points perpendicular to the surface of the ground we can calculate the roll and pitch angles. Therefore we simply analyze the contribution of the graviational acceleration to the X, Y and Z axis. See figure 2.5.

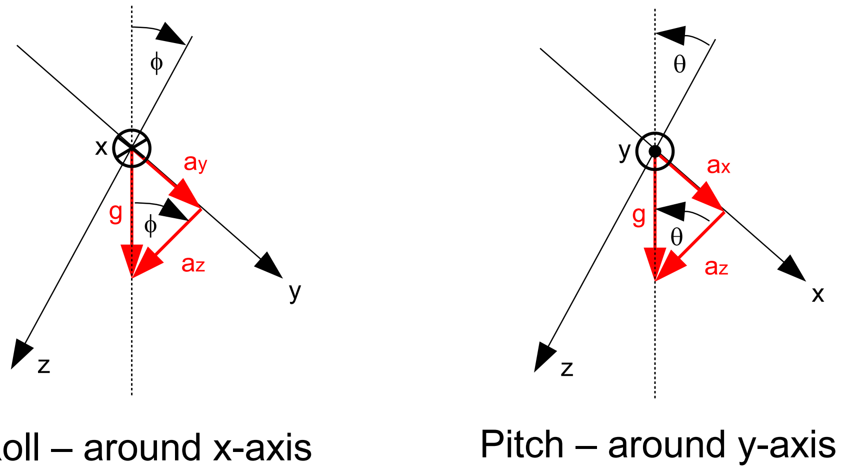

Figure 2.5: Determine Angles Using Accelerometer

The roll and pitch angles ϕ and θ can be calculated as follows.

$$\phi = \arctan \frac{a_y}{a_z} \tag{2.4}$$

$$\theta = -\arctan \frac{a_x}{a_z} \tag{2.5}$$

Calculating the RP angles in this way has the advantage that no drift of the angles can occurr. But the accelerometer reacts slower to changes and is very sensitve to vibration noise. When the motors are on then there's so much noise that you cannot determine any angle at all (see figure 2.4).

2.2.2 Kalman Filter: Sensor Fusion

As we have seen in section 2.2.1 we cannot use the aquired RPY angles right away for the attitude controller. There is either to much drift or to much noise. The solution to this problems is Sensor Fusion. Sensor Fusion means that we take the angle measurements acquired from the accelerometer and fuse them with the acquired angles from the gyroscope in such a way that we get a result which is better than

the measurement on their own.
Sensor Fusion can be done in two ways. Either you use an complementary filter or you use a Kalman Filter. Usually the Kalman Filter has a better performance than the complementary filter but it's much harder to find good parameters for the Kalman Filter.
I've decided to use a Kalman Filter to get the best possible result.
In order to operate correctly the Kalman Filter needs an predict function which is a model of the noisy system and an update function where the new system state is updated by using an additional sensor input.

Predict function As predict function I use a model for an integration system with drift. The model can be described as follows:

$$\phi' = \phi + \dot{\phi}\,\mathrm{d}t - \dot{\phi}_0\,\mathrm{d}t \tag{2.6}$$

- ϕ' is the new predicted angle
- ϕ is the old updated angle of the previous time step
- $\dot{\phi}\,\mathrm{d}t$ is the update using the gyroscope data
- $\dot{\phi}_0\,\mathrm{d}t$ is the drift angle

The system of that model is:

$$\begin{bmatrix}\phi' \\ \dot{\phi}_0'\end{bmatrix} = \begin{bmatrix}1 & -\mathrm{d}t \\ 0 & 1\end{bmatrix}\begin{bmatrix}\phi \\ \dot{\phi}_0\end{bmatrix} + \begin{bmatrix}\mathrm{d}t \\ 0\end{bmatrix}\dot{\phi} \tag{2.7}$$

Update function For the update function I use the angle which is aquired by using the accelerometer output. The error between prediction and sensor output can be calculated as follows:

$$y_0 = \begin{bmatrix}\phi_{accel.} \\ 0\end{bmatrix} - \begin{bmatrix}1 & 0 \\ 0 & 0\end{bmatrix}\begin{bmatrix}\phi' \\ \dot{\phi}_0'\end{bmatrix} \tag{2.8}$$

The system is then updated in such a way that the error y_0 is quadratically minimized. After performing the update we get the best estimation for system state which is the angle ϕ'' and the drift $\dot{\phi}_0''\,\mathrm{d}t$.

Results As it can be seen in figure 2.6 the estimated angle of the Kalman Filter is clearly better than the two sensor outputs on their own. The estimated angle has no drift and nearly no noise. It also reacts faster to changes than the accelerometer angle output.

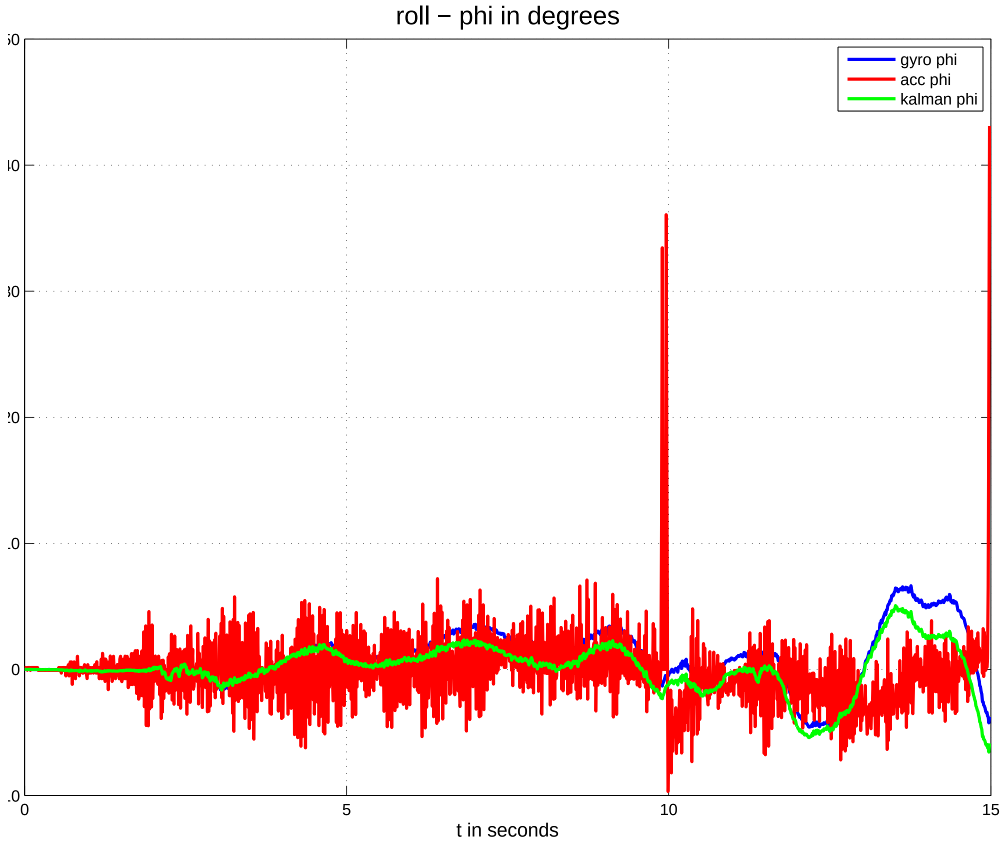

Figure 2.6: Roll Angle Complete Measurement

2.2.3 Attitude Controller

Each of the four DOFs (RPYH) gets its own PID controller. See figure 2.7. The controller is fed with estimated roll and pitch angles of the Kalman Filter, with the yaw angle of the integrator and the acquired height of the ultrasonic altitude sensor.

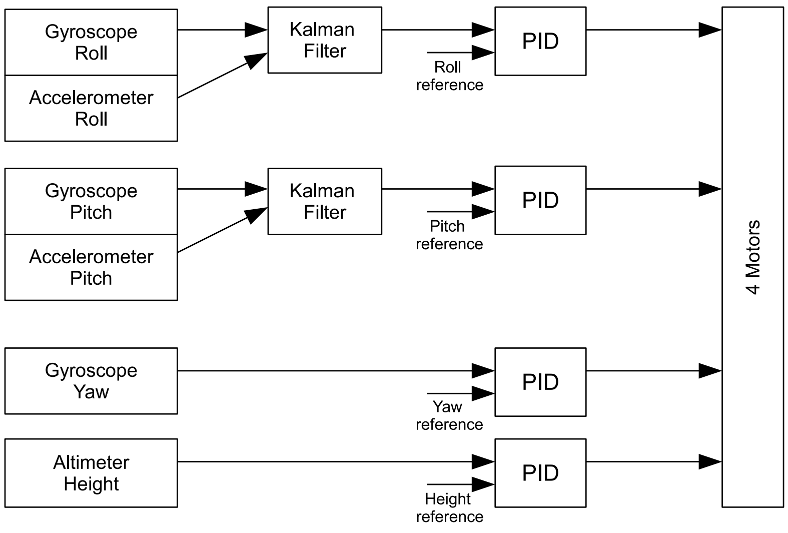

Figure 2.7: Attitude Controller

The parameters of the PID controllers are dermined by trail and error.
Each output of the 4 PID controllers is connected to all the four motors of the drone. The output of a PID controller counts for a PWM value in percent. Depending on to which angle or respectivly height the PID controller belongs its output value is either added or substracted to the final PWM value for the motor. The choice of signs depends on how the drone has to react to changes in the input of the PID controller. For example if the drone has a displacement in the pitch angle the two front motors (m0 and m1) must generate either more or less thrust than the two back motors (m2 and m3). See figure 2.8. Such reflections must also be made for the remaining three DOFs. The result can be seen in table 2.1 and figure 2.9.

	ϕ_{PWM}	θ_{PWM}	ψ_{PWM}	h_{PWM}
motor 0	+	-	+	-
motor 1	-	-	-	-
motor 2	-	+	+	-
motor 3	+	+	-	-

Table 2.1: Signs for PID controller output connections to certain motor

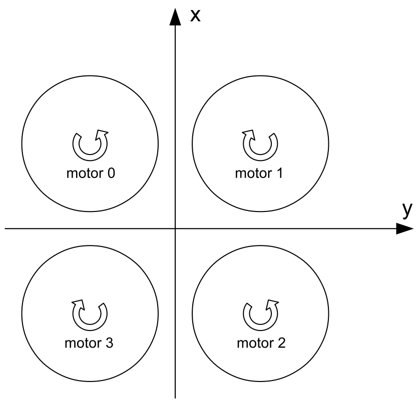

Figure 2.8: ARDrone Motors

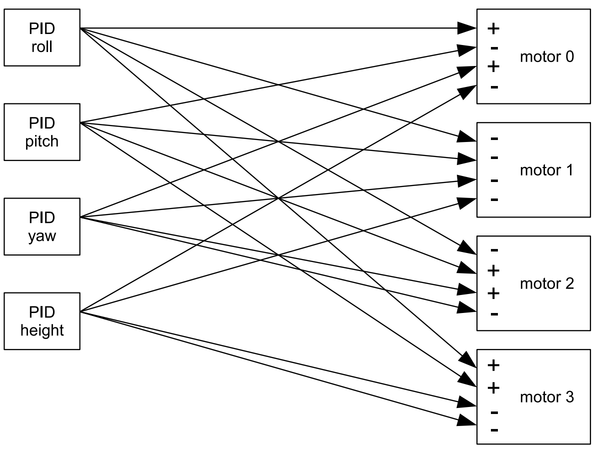

Figure 2.9: PID Connection Overview

2.2.4 Results

Test flights and figure 2.10 show that the new attitude controller shows a sufficiently good controlling behaviour. The new attitude controller enables the drone to fly in a stable manner.
The test flights also show that the drone is not able to hover without horizontal drift. As the attitude controller only controls RPYH there is no way to influence the horizontal drift.
To detect the drift additional sensors are needed. In the next two chapters I describe how eDVS sensors can be used in combination with a fast Optic Flow algorithm to control and minimize the horizontal drift of the drone.

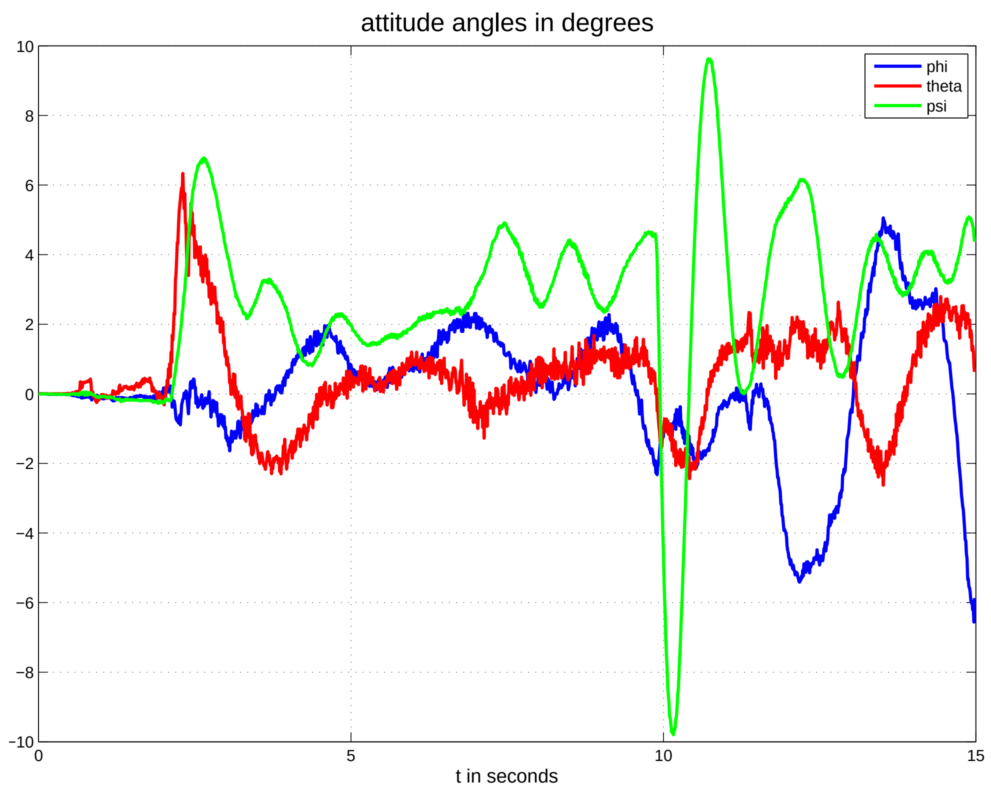

Figure 2.10: Controlled System Behaviour (Hovering Drone)

Chapter 3

eDVS Sensors and Optic Flow

In this chapter I show how two eDVS sensors can be connected directly to the AR.Drone mainboard. I also explain what modifications on the existing Optic Flow algorithm must be done so that the algorithm runs sufficiently fast on an embedded system with limited computing power.

3.1 eDVS Sensor Integration

Before I explain how the eDVS sensors are connected to the motherboard of the drone I want to show some properties of the eDVS sensors.

eDVS sensor properties eDVS sensors transmit every signle event over an UART connection with RTS/CTS hardware handshaking. The standard UART baudrate is 4 MegaBaud.
The eDVS sensor has an active region of 128x128 px. Whenever a sufficiently large intensity change occurrs in one pixel the sensor generates an event. Such an event consists of pixel coordinates and a polarity bit. The pixel coordinates hold the event location on the sensor and the polarity bit defines whether the event is an ON event or an OFF event. ON events imply raising light intensities and OFF events falling light intensities.
On very vivid scenes one eDVS sensor usually sends about 100000 events/s.

Connection to AR.Drone An overview about how the two eDVS sensors are connected to the AR.Drone can be found in figure 1.1 of chapter 1 and in figure 4.2 of chapter 4.
In order to connect the two sensors to the drone a FTDI serial to USB converter with two serial ports (two UART ports) is needed. Therefore I designed a new PCB. The two eDVS sensor are connected via UART to that adapter board while the adapter board itsself is connected to the drone via USB.
To access the serial to USB converter on the embedded system I have to cross com-

pile the USB driver. After loading the driver modules into the linux kernel the two eDVS sensors can be accessed via virtual serial ports. The virtual serial ports have the linux device files "ttyUSB0" and "ttyUSB1".
Unfortunatelly the drone only supports full speed USB devices. In that mode the serial to USB converter only supports a maximum of 3 MBaud per UART channel. This speed is not supported by the eDVS sensor. Therefore 2 eDVS are reprogrammed and the connection speed of the UARTs are set to 2 MBaud. Fortunatelly the speed is still fast enough so that hardly any event is lost.

3.2 Speed Optimization on Optic Flow Algorithm

The optic flow algorithm uses events captured by the eDVS sensors as inputs and computes global movements across the active region of the sensors. See figure 3.1 and 3.3.

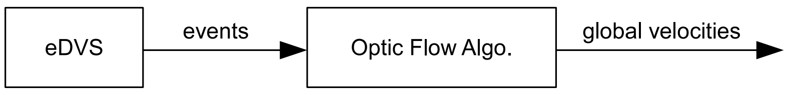

Figure 3.1: Optic Flow Overview

First the algorithm uses the events to detect local velocities. Therefore the algorithm analyzes the neighbors of the current event. Only neighbors with the same event polarity are analized. The analizing of neighbors implies that time stamps (absolute arrival time of the events) of the neighbors are compared to time stamp of the current event. See figure 3.2.

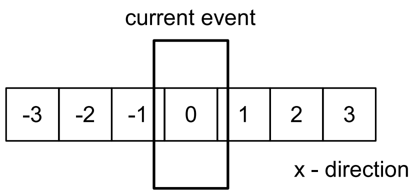

Figure 3.2: Event Neighbors

This is done for neighbors in the x-direction and neighbors in the y-direction. The smallest of all time differences between neighbors and current event is remembered. Whenever this time difference is smaller than the time difference between current event and last event then a new local speed is calculated.

Let's say an event occurred in location (x, y) with a time stamp $t_{x,y}$. The last time stamp of an event at the same location was $t'_{x,y}$. When analyzing the x-direction the neighbors $\{..., (x-2, y), (x-1, y), (x+1, y), (x+2, y), ...\}$ are processed. The time differences are:

$$\begin{aligned} &\vdots \\ \Delta t_{x-2,y} &= t_{x,y} - t_{x-2,y} \\ \Delta t_{x-1,y} &= t_{x,y} - t_{x-1,y} \\ \Delta t_{x+1,y} &= t_{x,y} - t_{x+1,y} \\ \Delta t_{x+2,y} &= t_{x,y} - t_{x+2,y} \\ &\vdots \end{aligned} \tag{3.1}$$

When the minimum time difference $\min_{i \neq 0} \Delta t_{x+i,y}$ is smaller than $\Delta t_{x,y} = t_{x,y} - t'_{x,y}$ than a new local velocity is calculated.

$$v_{local} = \frac{i_{min}}{\min_{i \neq 0} \Delta t_{x+i,y} - \Delta t_{x,y}} \tag{3.2}$$

i_{min} is the index of the neighbor with the minimum difference $\Delta t_{x+i,y}$.
The global velocity can be determined by using local velocities. See figure 3.3.

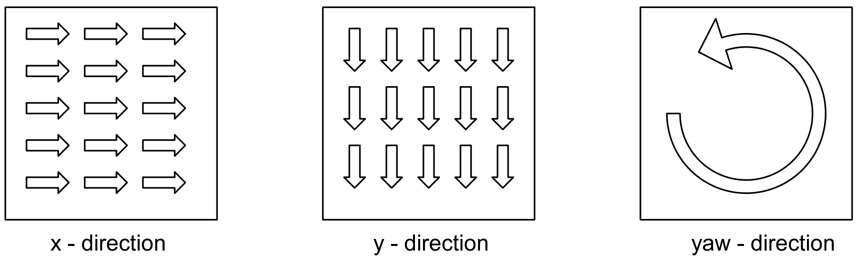

Figure 3.3: Global Velocities

The existing Optic Flow Algorithm All the things which are mentioned above are already realized in a C++ Optic Flow class. However tests of this algorithm on the embedded system of the drone show that the current implementation is far too slow. The embedded system has not sufficient computing power. Only about 20000 events per second can be processed. The current implementation is so slow because

the ARM9 CPU of the drone has no floating point unit (FPU). So the embedded system must emulate floating point operations in software what is really slow. One floating point multiplication needs about 1000 CPU cycles.
To sum up the current implementation of the Optic Flow algorithm must be optimized for the embedded system. The main optimization task is to avoid floating point calculations wherever it is possible.

Speed optimization The optimization task is to speed up the Optic Flow calculations that at least 200000 events per second for two eDVS sensor inputs can be processed on the existing embedded system of the drone. This implies a needed speed up by a factor of at least 10.

There are several optimization possibilies:

- avoid floating point operations
- use shift operators instead of integer multiplications / divisions
- calculate only data which is needed
- calculate same data only once
- introduce early calculation termination by adding if-clauses
- process chunk of events rather then calling a function for each single event

Results Tests of the optimized Optic Flow algorithm on the embedded system show that the algorithm can process now about 400000 events per second. This is definitively fast enough to process data from two eDVS sensors.

Chapter 4

Horizontal Drift Compensation Controller

The perpose of this chapter is to show how to design a drift compensation controller which uses as inputs velocities estimated by the Optic Flow Algorithm.
Before I start with the controller design I want to find out the relations between optic flow velocities and drone attitude. The drone attitude is now extended from a RPYH representation to a RPYHXY presentation. X and Y represent horizontal coordinates of drone with respect to the ground plane. See figure 4.1.

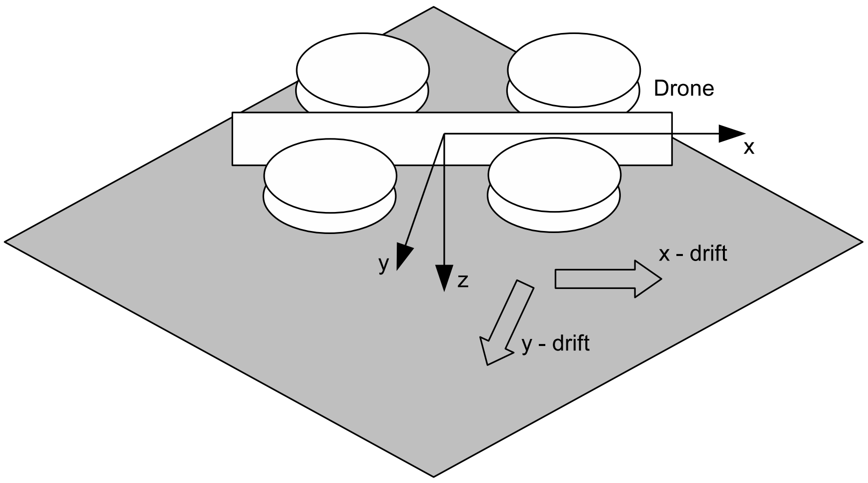

Figure 4.1: Horizontal Drifts

4.1 Relation between Optic Flow and Attitude

The relations between Optic Flow velocities and the drone attitude RPYHXY can be found by inspecting figure 4.2. Note that all velocities estimated by the Optic Flow algorithm are velocities of objects which move with repect to the eDVS sensors. This means when CAM1 which points to the ground dectects movements this movements has to be interpreted as movements of the ground while the drone stays on a fixed place. As a result to get the velocities of the drone the signs of the calculated drift speeds must be negated.

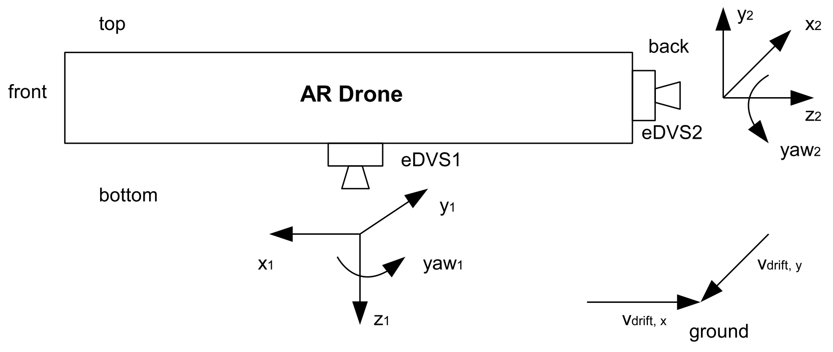

Figure 4.2: eDVS on ARDrone

The inspection of figure 4.2 results in the following equations:

$$v_{x,1} = v_{drift,x} - \dot{\theta} \qquad v_{x,2} = v_{drift,y} + \dot{\psi} \tag{4.1}$$

$$v_{y,1} = v_{drift,y} + \dot{\phi} \qquad v_{y,2} = -\dot{h} + \dot{\theta} \tag{4.2}$$

$$v_{yaw,1} = \dot{\psi} \qquad v_{yaw,2} = -\dot{\phi} \tag{4.3}$$

These equations can be used to calculate the drift of the ground $v_{drift,x}$ and $v_{drift,y}$:

$$v_{drift,x} = v_{x,1} + \dot{\theta} = v_{x,1} + (v_{y,2} - \dot{h}) \tag{4.4}$$

$$v_{drift,y} = v_{y,1} - \dot{\phi} = v_{y,1} - v_{yaw,2} \tag{4.5}$$

$$v_{drift,y} = v_{x,2} - \dot{\psi} = v_{x,2} - v_{yaw,1} \tag{4.6}$$

But the equations can also be used to calculate RPYH:

$$\dot{\phi} = v_{y,1} - v_{drift,y} = -v_{yaw,2} \tag{4.7}$$

$$\dot{\theta} = v_{drift,x} - v_{x,1} = v_{y,2} - \dot{h} \tag{4.8}$$

$$\dot{\psi} = v_{yaw,1} \tag{4.9}$$

$$\dot{h} = -v_{y,2} + \dot{\theta} \tag{4.10}$$

Under the assumption that the drone is controlled by the RPYH attitude controller the velocities of these 4 DOFs are zero. Consequently the horizontal drift $v_{drone,drift,x}$ and $v_{drone,drift,y}$ can be calculated as follows:

$$v_{drone,drift,x} = -v_{drift,x} = -v_{x,1} \tag{4.11}$$
$$v_{drone,drift,y} = -v_{drift,y} = -v_{y,1} \tag{4.12}$$

These results can be used by the horizontal drift compensation controller.

4.2 Drift Compensation Controller

The perpose of the drift compensation controller is to minimize the horizontal drift of the drone. As I've already mentioned before the RPYH attitude controller cannot do that.
The idea for realizing such a drift compensation controller is to use the horizontal drifts as inputs and then calculate adjustments on the RP angles to counter these drifts. For example a positive drift into the x-direction can be countered by setting a pitch angle θ bigger than zero. To counter positive drift into the y-direction the roll angle ϕ can be set smaller than zero.
The final controller can be seen in figure 4.3.

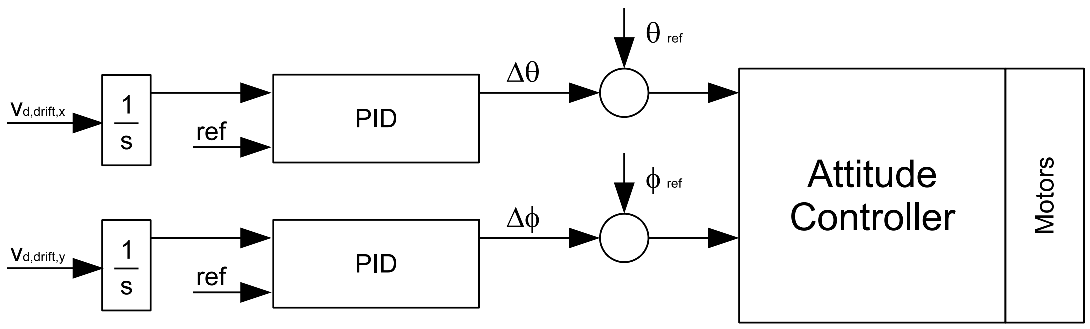

Figure 4.3: Drift Compensation Controller

4.3 Results

Unfortnatelly I wasn't able to verify the performance of the drift compensation controller. When I try to run the RPYH attitude controller and the optic flow algorithm at the same time the motors of the drone suddenly stop.

Further investigations show that loading the USB-OTG driver module into the embedded linux kernel disturbs the motors. As it can be seen in figure 4.4 loading the USB OTG driver causes a voltage drop in the 5V supply line. That 5V supply line also supplies the motors and a voltage drop disturbs them.
After being disturbed by the voltage drop the motors can only be restarted after a switch off and on of the AR.Drone.

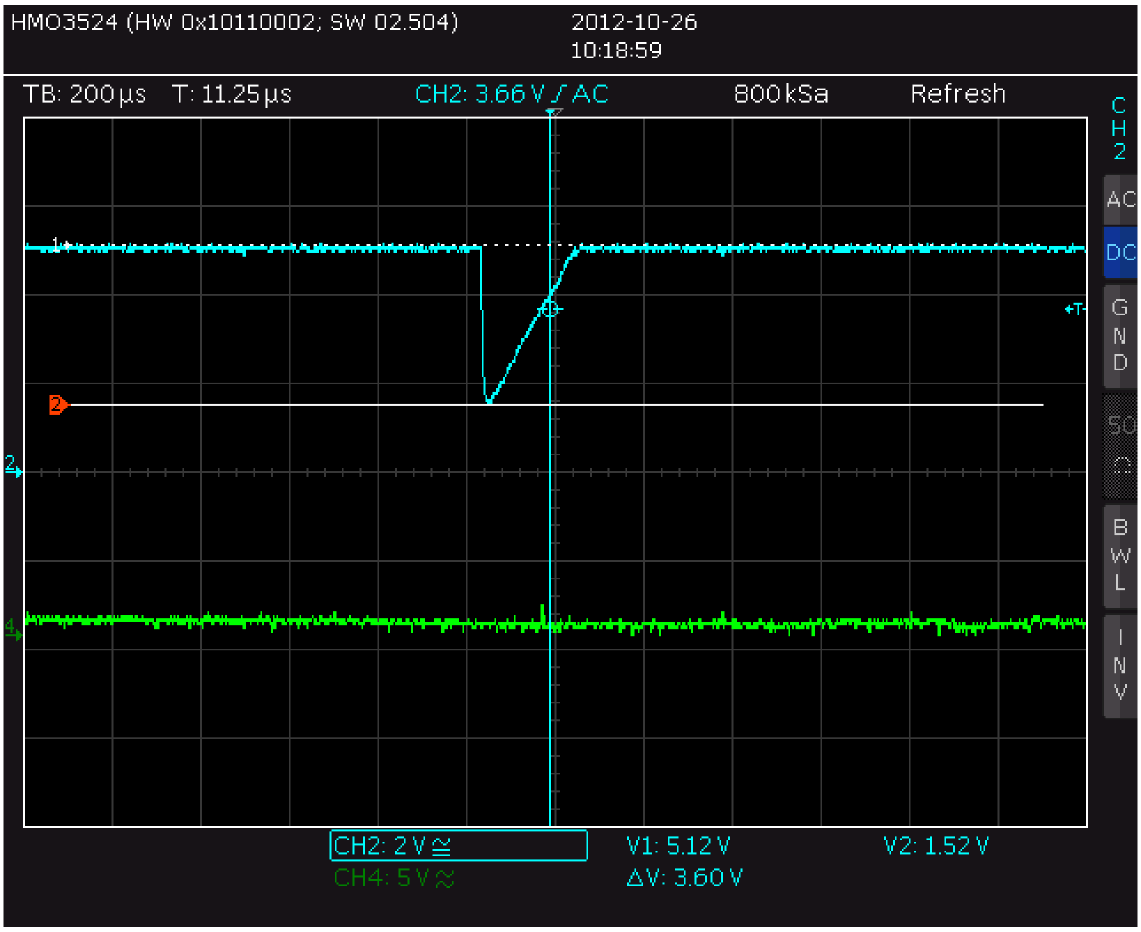

Figure 4.4: Voltage drop in 5V supply line

4.4 Outlook

It has been shown that all the Optic Flow calculations can be done on an embedded system alone. Unfortunatelly it wasn't possible to verify the performance of the drift compensation controller. The USB-OTG driver problem couldn't be solved in time.
Nevertheless I think the driver problem can be solved by debugging the driver. I also think that the drift compensation controller will show a good performance. As

it is directly connected the attitude controller it will have a very tight grip on the drone.

List of Figures

Bibliography

[1] Pflaum, A. and F. Eutermoser: *Quadrocopter Stabilization using Neuromorphic Embedded Dynamic Vision Sensors (eDVS)*. NST, TUM, 2012. Practical Course.

[2] Perquin, Hugo: *Tech Toy Hacks*. Online: http://blog.perquin.com/.

[3] Miller, J. M. and D. A. Robinson: *The navigation and control technology inside the ar.drone micro uav*. Preprints of the 18th IFAC World Congress, September 2011.

[4] *File: RPY angles of airplanes.png*. Online: http://en.wikipedia.org/wiki/ File:RPY_angles_of_airplanes.png.

[5] Conradt, J., Tevatia, G., Vijayakumar, S., & Schaal, S. (2000). On-line Learning for Humanoid Robot Systems, International Conference on Machine Learning (ICML2000), Stanford, USA.