Controlling multiple drones autonomously inspired by birds ability to keep formation

University:
University of Southern Denmark

Author:
Mathias Mikkel Neerup

Bachelor Thesis

In coorperation with:
Jussi Hermansen
Info@viacopter.eu
Cottagevej 4
3300 Frederiksvaerk

Supervisor:
Kjeld Jensen
kjen@mmmi.sdu.dk
Maersk Mc-Kinney Moeller
University of Southern Denmark

June 1, 2016

Mathias: : A bachelors thesis (15 ECTS) report from a single student should have a length no more than 35 pages plus maximum 10 pages of appendices.

Mathias: : Ja, men gør det som elektronisk bilag (pdf) og beskriv i rapporten, at den ligger der. Marker det evt. dine bidrag med farve, så det er meget let at se.

Among other things...
Project title
Project period
Your name, email address etc.
Report type (e.g. Bachelor thesis)
Your education (e.g. Robotics)
"University of Southern Denmark" and name of center, department or faculty. Names of supervisor(s)
Copyright and license statement for report and material in the electronic archive

Todo list

Abstract

Until now drones keeps getting bigger and larger to carry bigger batteries with more capacity and to lift heavier payloads. This leads to drones getting less efficient, less responsive and gets more dangerous. Instead it has become popular to make drones smaller and increase the number of drones needed to solve a task.

Materials & methods
This thesis describes how to make three drones follow a leader drone with a preprogrammed path as an example of drones cooperating. A Linux PC running MarkerLocator tracks each drones position and wirelessly transmits, using Xbee, the drones positions to all drones.The position of each drone is spoofed into the drone using the CAN-bus and thereby overwriting the onboard GPS. An outdoor test has been made using the onboard GPS to test the leader-follower algorithm in a bigger scale. A small PCB has been developed and mounted on each drone to route packages from the Xbee module to the CAN-bus of the drone and to measure the local altitude of the drone using a ultrasonic sensor. The PCB carries an AT90CAN128 as microcontroller which build-in CAN support making it obsolete to carry an external USB CAN-controller.

Results -¿ discussion
The accuracy of the vision based localisation is measured using a laser pointer pointing out the drones 2D position on the floor making it possible to measure the variance of the drones position. The leader-follower algorithm was also tested outside using the onboard GPS. The performance of the leader-follower algorithm is measured and discussed using plots that reveals the distance between the drones.

Conclusion -¿ perspective
It is shown that it is possible to implement the leader-follower algorithm using a vision and ultrasonic based positioning system. The distance between all drones when flying indoor was +/- 10 cm which is less than the maximum accepted error. It was possible to add 5 follower-drones without editing the code showing it is a generic system The system can further be used to indoor testing of navigation algorithms and explore the many possibilities drones has to offer. If drones at some point needs to fly indoor to help eg. Mobile robots navigating, vision might be a way to obtain a absolute indoor position for the drones.

Resumé

abstract på dansk

Reading Guide

List of Abbreviations

AQ	AutoQuad Flight Pilot

Preface

Thanks to:

- Kjeld Jensen
- Henrik Midtiby
- Morten Albeck Nielsen
- Anna Riisberg Soerensen
- Mads Tilgaard Jensen & Eskild Andresen

Table of Contents

Chapter 1

Introduction

Drones are more and more used in different applications in different areas because they make it relatively easy to get a quick or more in depth overview of a situation. A less positive thing of the drones is the amount of energy they are capable of carrying. A drone designed with long time-flight flying in mind is capable of flying approximately 20 minutes depending on weather and payload. If the drone is equipped with a heavy camera or other kind of payload, then the flight time starts to decrease rapidly. So far the solution has been to increase the size of the drones to mount bigger batteries, but this might not be right way of doing it. Drones become less efficient, less responsive and gets more dangerous due to increase in weight.

By looking at the nature, one can easily see how small animals like ants and birds manage to cooperate and thereby build or move bigger things that they would not be able to do on their own. This way of small independent, decentralized units working together is called a swarm.

By making drones smaller, they get more efficient, their flight time increase and they get cheaper but of the cost of their ability to lift. Therefore it seems like an idea to make small drones cooperate to solve more complex tasks.

SDU is currently using a platform called AutoQuad which is supported by large and small drones. Solving a task as a swarm is complex, and is not in the scope of this project. This project intend to implement a Leader-follower principle inspired by birds flocking behaviour. The leader-drone will have a preprogrammed flight path and the follower-drones will have no knowledge about the flightpath. Each follow-drone will try to keep a distance of 50 cm within plus minus 10 cm to its neighbours and the leader-drone. A computer program will be written to control each drone and tell them where to go without colliding. This will be developed in an indoor controlled environment. Since GPS is unavailable indoor, this project will be using vision, e.g. Henrik Midtiby's MarkerLocator to get the location of each drone. When the Follow-leader has been implemented it should in theory work outside as well, though with larger distances due to GPS inaccuracy.

Mathias: Introduction: Udvide Autoquad så man(SDU) kan spænde en RTK gps på og styre dronen inden for få cm

1.0.1 Related Work

1.0.2 Problem Statement

The current AutoQuad does not support vision as source of 2D position which is required for the indoor Leader-follower to work. The relative height of the drone is measured using the build-in barometer in each drone providing the third dimension to the drones position. In

case the barometer turns out to be too inaccurate due to drift other sensors might be used e.g ultrasound. The computer has to send waypoints wirelessly to all of the drones. A PCB for each drone has to be developed for the drones to carry as payload. The PCB will be responsible for receiving messages from the computer and translate them into the CAN-bus of the drone. Hypothesis If each drone's 2D position is obtained using vision and spoofed into the drone using CAN, then it is possible for at least 3 drones to follow a leader drone with a preprogrammed flight path and keep a euclidean distance at 50 cm within plus minus 10 cm to the leader and its neighbours.

1.0.3 Hypothesis

If each drone's 2D position is obtained using vision and spoofed into the drone using CAN, then it is possible for at least 3 drones to follow a leader drone with a preprogrammed flight path and keep a euclidean distance at 50 cm within plus minus 10 cm to the leader and its neighbours.

Chapter 2

Materials and methods

This chapter concerns the most important parts about how this project has been implemented. It has been split into several sections.

Mathias: Materials and methods: Write shortly about the implementation Chapter

Mathias: Materials and methods: Less calculations on the drone, better to move to another computer/cluster with more power than the drone has available from its battery

Mathias: Materials and methods: Image of camera and drone seen from the side. Show what happens to the drones position estimation when it drifts down/up

2.1 CAN implementation

2.1.1 Introduction

In order to spoof the onboard GPS, the CAN-bus was chosen as input to AQ. The CAN-bus is already implemented and widely used on a ViaCopter drone to communicate between AQ and hardware like ESCs, PDB etc. Much time was spend reading through the source code and trying to figure out how the protocol works. The developers behind AQ says, that the code is the documentation and thereby have not written any real documentation.
Figuring out how it works was done using debugging utilities such as breakpoints and looking at the content of the different memory locations in CrossWorks.
More debugging tricks can be seen in appendix **??**. To further investigate the protocol a PEAK-CAN adapter were connected to a Ladybird drone and several python scripts were developed to send and receive messages from AQ.
The author did not go into details about messages used between AQ $\leftrightarrow$ ESC's and AQ $\leftrightarrow$ OSD.

2.1.2 CAN Protocol description

A Peak-CAN adapter were used throughout the project. It supports High-speed ISO 11898-2[1] which is a standard that states the properties of physical layer. Its max speed is 1 mbit which is the speed AQ uses to communicate on the CAN-bus. AQ uses CAN 2.0B which has 29 identifier bits. AQ is not using an existing protocol on top of ISO 11898-2. The developers created their own to suit their needs.

[1] http://www.peak-system.com/PCAN-USB.199.0.html?&L=1

Identifier bits

A Generic look at a AQ CAN messages can be seen in table 2.1. The normal function of the identifier bits is the priority of the messages. A CAN controller can then be setup to only allow certain messages to be processed depending on their identifier bits. The identification bits has been split up in AQ to contain information about the sender, receiver etc. Which is not normally saved information in a CAN message.

CAN Message							
Identfier bits [29 bits]							Data bits [max 64 bits]
LCC [28:27]	TT [26]	FID [25:22]	DOC [21:16]	SOID [15:11]	TID [10:16]	SEID [5:0]	

Table 2.1 *Table shows the identifier bits used in AutoQuad CAN messages*

In figure 2.2 the abbreviations can be seen.

LCC	Logical Communications Channe
TT	Target Type
FID	Function ID
DOC	Data Object Code
SOID	Source ID
TID	Target ID
SEID	Sequence ID

Table 2.2 *Table shows abbreviations used in table 2.1*

Each of the elements in an AQ messages will be explained how they are used in AQ.

Logic Communication Channel

LCC is the priority of the message. To understand how LCC works, one needs to look into CAN arbitration. As stated in ISO-11898-2, a 0 is the dominant bit and a 1 is the recessive bit. The example in figure 2.1 shows two nodes each transmitting a packet. The arbitration only happens during the transmission of the identifier bits. In the example on bit 8, Node 16 losses the arbitration and stops transmitting. Node 15 keeps transmitting its packet because it has lower identifier bits and thereby higher priority.

	Start Bit	ID Bits											The Rest of the Frame
		10	9	8	7	6	5	4	3	2	1	0	
Node 15	0	0	0	0	0	0	0	0	1	1	1	1	
Node 16	0	0	0	0	0	0	0	1	Stopped Transmitting				
CAN Data	0	0	0	0	0	0	0	0	1	1	1	1	

Figure 2.1 *Example of CAN arbitration where node 15 has the lowest ID and thereby highest priority*

LLC in a message can take one of the defines in code 2.1

Listing 2.1 LLC defines

```
#define CAN_LCC_EXCEPTION ((uint32_t)0x0<<30)
#define CAN_LCC_HIGH      ((uint32_t)0x1<<30)
#define CAN_LCC_NORMAL    ((uint32_t)0x2<<30)
#define CAN_LCC_INFO      ((uint32_t)0x3<<30)
```

Table 2.3 *Table showing the 4 types of priority in AQ*

Name	Value	Priority (lowest first)
EXCEPTION	0000	0
HIGH	0001	1
NORMAL	0010	2
INFO	0011	3

Target Type

Target type can either be "node" or "group" depending on if the receiver(s) is one or more nodes.

Listing 2.2 Target type defined in AQ

```
#define CAN_TT_GROUP      ((uint32_t)0x0<<29)
#define CAN_TT_NODE       ((uint32_t)0x1<<29)
```

The GROUP is used when AQ sends its reset-msg upon startup.
When the authors node sends a sensor-measure to AQ it will be of type CAN_TT_NODE.

Function ID

Function ID describes the function of the CAN message. If it is a PING, ACK, NACK, etc. It simply states the function of packet so the receiver node knows what to do with the message. Different types of functions can be seen in code 2.3

Listing 2.3 Excerpts from AQ's list of function defines

```
#define CAN_FID_RESET_BUS  ((uint32_t)0x0<<25)
#define CAN_FID_ACK        ((uint32_t)0x1<<25)
#define CAN_FID_REQ_ADDR   ((uint32_t)0x7<<25)
#define CAN_FID_GRANT_ADDR ((uint32_t)0x8<<25)
#define CAN_FID_CMD        ((uint32_t)0x3<<25)
```

Table 2.4 describes the FIDs shown in listing 2.3.

Mathias: Function ID: Dette er forkert, eftersom alle defines er taget fra AQ's can.h så der passer det..

If a message is if FID CAN_FID_CMD, the Data Object code will be used as the command. It should be noted from code 2.3 that the FID's is moved 25 positions to the left but FID is

shown in table 2.1 is at bits 22:25. The offset of three bits is due to the way ARM's CAN registers is implemented. The three initial bits in CAN_TIxR and CAN_RIxR (registers used to transmit and receive respectively) [2] registers are general information about the CAN messages like whether it is extended (Part B.0) or not.

Table 2.4 *Descriptions of the FIDs mentioned in code 2.3*

FID	Description
CAN_FID_RESET_BUS	Message sent by AQ when powered up
CAN_FID_ACK	Message sent to AQ when acknowledging a received message
CAN_FID_REQ_ADDR	Messaged used when a node on the bus tries to register itself as a node on the bus
CAN_FID_GRANT_ADDR	Messaged sent by AQ when it registers a node as a node on the bus.

Table 2.4 shows a description of the 4 FIDs used when a node is registered.

Data Object Code

Data Object code is used as a parameter to the FID. Data Object Code can have different values depending on the fid. Code 2.4 shows a function using DOC when message FID is CAN_FID_CMD.

Listing 2.4 Snippet of AQ's can.c:365

```
1  void canProcessCmd(... ,uint8_t doc, ...) {
2      switch (doc) {
3          case CAN_CMD_STREAM:
4           ...
5              break;
6          case CAN_CMD_TELEM_VALUE:
7           ...
8              break;
9          case CAN_CMD_TELEM_RATE:
10          ...
11             break;
12     }
13 }
```

In section 2.1.3 DOC is used by the author to tell which part of the GPS data is getting sent but where FID is CAN_FID_TELEM.

Mathias: Data Object Code: Fixe ref til section om GPS spoof

Source/Target/Network ID

When referred to source/target ID but without respect to either sender or receiver but just a CAN-node, then it is referred to as NetworkId. When a node registers itself in AQ, it gets assigned a NetworkId.

[2] http://www2.st.com/content/ccc/resource/technical/document/reference_manual/59/b9/ba/7f/11/af/43/d5/CD00171190.pdf/files/CD00171190.pdf/jcr:content/translations/en.CD00171190.pdf:688

Sequence ID

The sequence id is incremented on transmission of each message. It is used when eg. A CAN-node sends a reply to a get-message, it is then including the sequence id of the get-message so AQ knows what it receives a reply for.

Data bits

The data bytes does not have a general definition, it depends on the function id. With some FIDs the data field may contain additional parameters.

Eg. When FID is CAN_FID_REQ_ADDR, data has the fields shown in table 2.5

Table 2.5 *Packet sent from node when registering in AQ*

64 bits							
0	0	CanId	CanType	UUID3	UUID2	UUID1	UUID0
7	6	5	4	3	2	1	0

UUID is a unique address generated by each node on the bus. In ESC32v2[3] the address is calculated using XXH-hashing algorithm. The algorithm generates a 32 bit hash from a given input value and a salt. The input to XXH in ESC32v2 is the unique ID that every ARM in the STMF32[4] family has. Code 2.5 shows how it is implemented in AQ.

Listing 2.5 Snippet showing UUID generated in ESC32v2

```
1 can.h:20
2 #define CAN_UUID 0x1FFFF7E8
3
4 can.c:753
5 canData.uuid = XXH32((void *)CAN_UUID, 3*4, 0);
```

CanType is the type of the node trying to register. Eq. SENSOR, ESC and SERVO.

CanID is the number of the CanType node trying to register. When an ESC32v2 is trying to register it sends CanId as the ESC number ranging from 1 to the number of ESC's mounted on the drone. The CanID is assign to each ESC manually as part of the configuring.

2.1.3 Registering a node in AQ

When the AQ is starting up, it is transmitting a CAN_FID_RESET_BUS to the bus. When each CAN-node receives the messages, they send out a CAN_FID_REQ_ADDR message in order to get registered as a node in AQ. If a node on the bus does not get registered it will not be able to communicate with AQ.
When AQ receives an CAN_FID_REQ_ADDR, it sends back a CAN_FID_GRANT_ADDR to tell the node that the registration is successful and to assign the node its networkID. This id is used later to identify the node when it is transmitting a message or when AQ wants to send

[3] https://github.com/bn999/esc32/blob/master/onboard/can.c
[4] http://www2.st.com/content/ccc/resource/technical/document/reference_manual/59/b9/ba/7f/11/af/43/d5/CD00171190.pdf/files/CD00171190.pdf/jcr:content/translations/en.CD00171190.pdf

out a message to that specific node. The sequence diagram in figure 2.2 shows the described sequence.

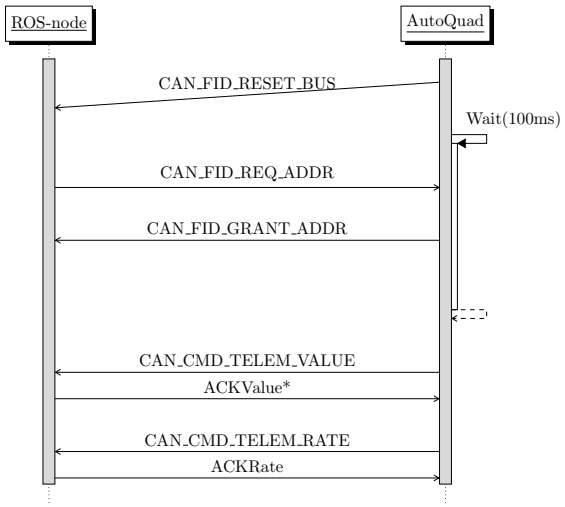

Figure 2.2 *Registering a new node in AQ*

Depending on the canType set in the data-field when a node is transmitting a CAN_FID_REQ_ADDR, AQ might send zero or more messages back to the node.
In figure 2.2 it can be seen that AQ sends back CAN_CMD_TELEM_VALUE and CAN_CMD_TELEM_RATE because the CanType is set to SENSOR. Each of the two messages has two fields in the data field, that contains information about the stream. CAN_CMD_TELEM_VALUE is used to request telemetry from a node that registered itself as a SENSOR. CAN_CMD_TELEM_RATE is the requested telemtry rate from AQ. Default is 10 hz [5].
A timeout timer is started when each message has been sent. If an ACK is not received after each message within 250 ms [6] a timeout counter is incremented. At the time of writing this information is not used but might be later on.

Code in 2.6 shows a snippet[7] from AQ where AQ waits for nodes to request an address.

Listing 2.6 Snippet showing AQ waits for devices

```
1 canResetBus();
2
3 // wait 100ms for nodes to report in
4 micros = timerMicros() + 100000;
5 while (timerMicros() < micros)
6     // extend wait period if more nodes found
7     if (canCheckMessage(0))
8         micros = timerMicros() + 100000;
```

AQ waits default 100 ms for nodes to request an address. If canProcessMessage() returns true a node tried to register and the registration period is extended 100 ms.

[5] https://github.com/bn999/autoquad/blob/master/onboard/canSensors.h:24
[6] https://github.com/bn999/autoquad/blob/master/onboard/can.h:71
[7] https://github.com/bn999/autoquad/blob/master/onboard/can.c:547

2.1.4 Registering node test

Implementation of a CAN-node running on a PC was initially done in python. The python module *SocketCan* [8] supports PeakCAN adapter out of the box.

Code 2.7 shows a snippet of main.py.

Listing 2.7 Snippet showing AQ registration from python

```
# Create autoquad handle
autoquadNode = AutoQuadNode('can1', 'socketcan')

# Wait for reset msg transmitted by AQ.
autoquadNode.WaitForReset(timeout=1000);
print "Recevied readymsg"

# Register node, CanType as sensor and CanID as PDB ampere
msg = autoquadNode.RegisterNode(CAN_TYPE_SENSOR, CAN_SENSORS_PDB_BATA)

# Wait for telemetryTryValue
msg = autoquadNode.recv()
# Parse message
msg = CanMessage(msg)

# Print all values received, _str__(self) overwrited
print msg

# ACK TelemValue
autoquadNode.AnswerRequestTelemValue(msg)

# Wait for telemetryRate
msg = autoquadNode.recv()

# Parse message
msg = CanMessage(msg)

# ACK TelemRate
autoquadNode.AnswerRequestTelemRate(msg)
```

When a node has successfully registered, it is shown in QgroundStation as shown in figure 2.3. It can be seen 4 motors were connected to the CAN bus, but also that the spoofed sensor was registered.

[8] http://python-can.readthedocs.io/en/latest/socketcan_native.html

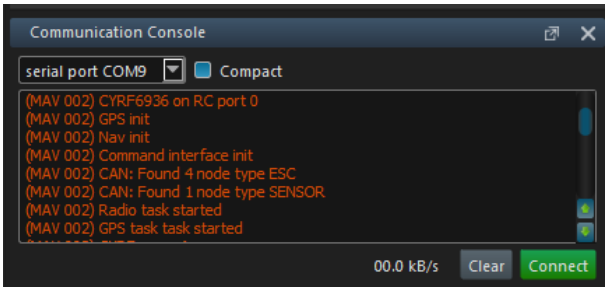

Figure 2.3 *Successful SENSOR registration*

2.1.5 Spoofed current test

When an AQ PDB is mounted on a drone, the PDB measures voltage, current and temperature and sends the measurements to AQ where they get logged.
The code in section 2.1.4 was further developed into a sensor simulator that spoofs current measurements into AQ to test the registering works probably.
A logger extension-board was mounted on the M4-board in order to save the measurements to a MicroSD-card.

Code 2.8 shows the implementation where a sinus is generated to simulate a current and how it is transmitted to AQ.[9]

Listing 2.8 Snippet spoofing current measurements into AQ

```
def ReqistrerTelem(self):
    # self.sinus is a list of sinus values
    sensor_value = self.sinus[self.phase]

    # Get next value next time and do overrun
    self.phase+=1
    self.phase = self.phase % 20

    # Pack as IEEE754 float
    sensor_value_float = pack('>f', sensor_value)

    # Create list of float-bytes as integers
    data_float = [int(ord(elm)) for elm in sensor_value_float]

    # Reverse list and append empty data
    data_float.reverse()

    # Send to AQ. id = CAN_FID_TELEM (02c00000) | Sourceid 1 (00000800) |
        CAN_SENSORS_PDB_BATA (00004000)
    self.send(0x02c04800, data_float)
```

The method defined in code 2.8 was run every 100ms.
After the code ran for a few seconds the SD card was plugged into a PC and QgroundStation configured to make a plot of the PDB_CURRENT-field. The plot can be seen in figure 2.4

[9]Some guy. 2993.

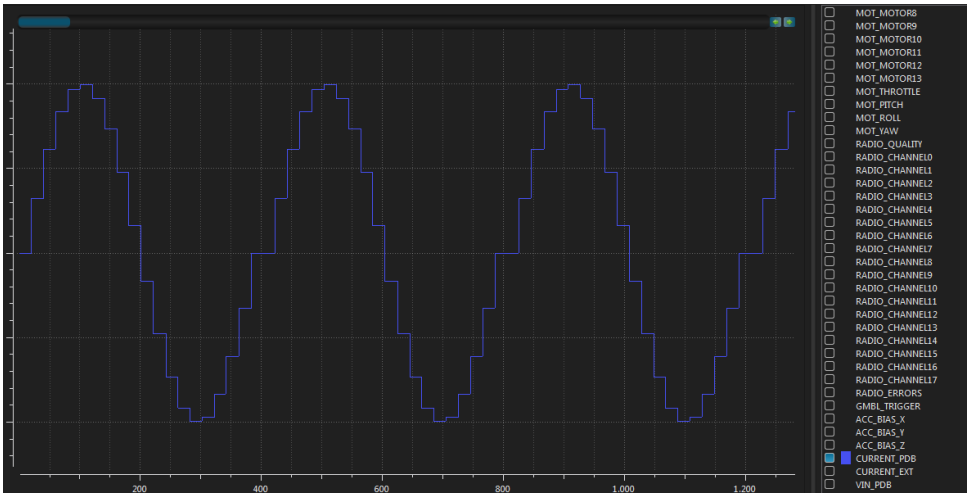

Figure 2.4 *Test of python CAN test*

Figure 2.4 shows the sinus generated from the node as expected. can0 00000000 [0] can0 01C00000 [8] EF BE AD DE 03 09 00 00 can0 0E000041 [6] EF BE AD DE 00 00 can0 14CD0042 [2] 00 08 can0 14CC0043 [2] 0A 00

seq tæller op så rate/value bliver acknowledged can0 00000000 [0] can0 01C00000 [8] EF BE AD DE 03 09 00 00 can0 0E000041 [6] EF BE AD DE 00 00 can0 14CD0042 [2] 00 08 can0 00400002 [8] 00 00 00 00 00 00 00 00 can0 14CC0043 [2] 0A 00 can0 00400003 [8] 00 00 00 00 00 00 00 00

2.2 Communication flow

Content	Lat	Lon	Height	DOP	CRC-16
Datatype	double	double	float	float	uint16_t
Bytes	31:24	23:16	15:8	7:4	3:0

Mathias: Communication flow: Ændre 0:3 til 0:1 da en uint16_t kun tager 2 bytes og ikke 4

Table 2.6 *Message sent from PC to ESP8266*

2.3 Vision based localization

- Introduction to section(chapter)
- Why use vision as lozalization. cheap, no expensive sensors.
- About the MarkerLocator(out of scope), how it works,(Overview - not in depth, why it's slow, how window-mode works) how it's implemented, scalability.
- Quality meassure, why is it nessecary.
- How I arrived to a good way of meassureing the found marker.
- One drone detection

- Two drones detection, different ways of detecting orders.
- Implementation in ROS, splitting into different nodes. optimization

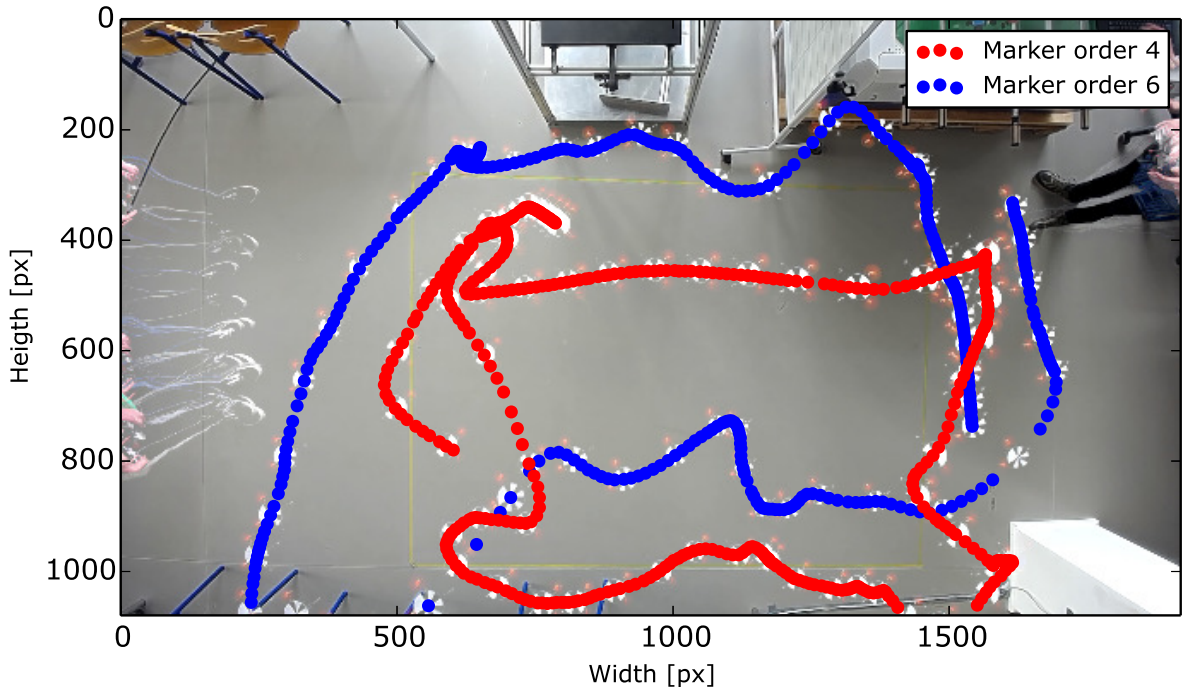

Figure 2.5 *Figure showing every 7'th frame merged into one frame combined with the tracked positions of the drones*

Mathias: Vision based localization: Smaller dot size - almost impossible to see the drone in the frames..
Mathias: Vision based localization: Red is above blue, not because the drone wasn't detcted
Mathias: Vision based localization: WIfi test distance, see what happens to latency

2.4 Software

2.4.1 Test of CAN, queues

In order to test CAN and queues, a task was written. The purpose of the task was to receive a CAN-message if any available from the queue and send the ID and data from the CAN message to a PC using UART. The task used can be seen in code 2.9.

Listing 2.9 Task used to test CAN

```
CAN_frame frame;
if(QueueReceive(&Queue_CAN_Rx, &frame)) {
   char ch;
   /* ID out uart0 */
   for(int i = 3; i >=0; i--){
      ch = ((frame.id >> i*8) & 0x000000FF);
      QueueSend(&Queue_Uart0_Tx,&ch);
   }
   /* ID end*/

   /* MSG out uart0 */
   for(int i = (frame.dlc-1); i >=0; i--){
      ch = ((frame.msg >> i*8) & 0x00000000000000FF);
      QueueSend(&Queue_Uart0_Tx,&ch);
   }
   /* MSG end */

   /* End of line */
   ch = '\r';
   QueueSend(&Queue_Uart0_Tx,&ch);
   ch = '\n';
   QueueSend(&Queue_Uart0_Tx,&ch);
}
```

The command used to send CAN-messages can be seen in code 2.10.

Listing 2.10 Task used to test CAN

```
while true; do sleep 1; print $(date); cansend can0 1DEADBEF#FEDCBA9876543210; done
```

The command sends a CAN-message using a connected PEAK-CAN adapter. It sends a message each second and prints out the date to make it clear when it is sending a message. *1deadbef* is the 29 bit ID and *fedcba9876543210* is the 64 bit data.

The command used to show the received HEX values can be seen in code 2.11

Listing 2.11 Command used to get UART messages

```
cat /dev/ttyUSB0 | xxd -c 14
```

Cat reads from /dev/ttyUSB0 and pipes its output to xxd where it is shown in HEX.
-c is number of columns which is 14 due to 4 ID bytes, 8 data bytes and 2 as newline.

The result can be seen in figure **??**.

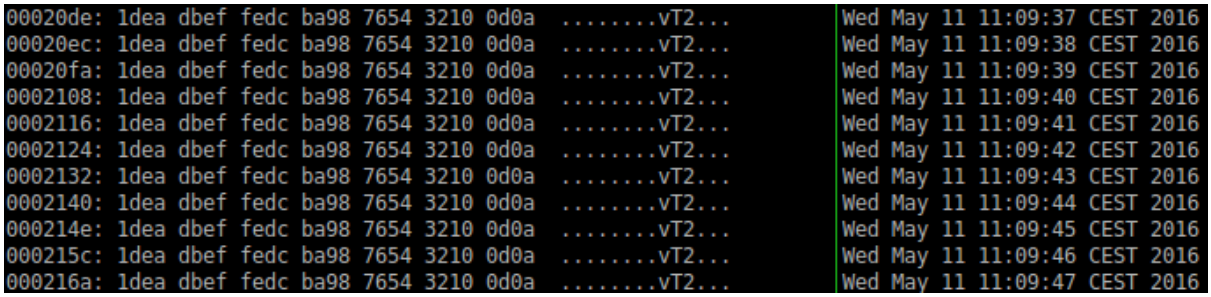

Figure 2.6 *Left shows output from command 2.11 and right shows command 2.10*

It can be seen that the received ID and data is *1deadbef* and *fedcba9876543210* respectively. *0d0a* is \r and \n respectively.

2.4.2 Test of RTCS timing

In order to test the timing of the scheduler, a led_task was written. The task can be seen in code 2.12

Listing 2.12 RTCS task used in timing test

```
1  void is_alive_task(uint8_t my_state){
2
3     // Write to UART0
4     UDR0 = my_state+'0';
5
6     switch(my_state){
7     case 0:
8        INT_LED_ON_GREEN;
9        INT_LED_OFF_RED;
10       INT_LED_OFF_BLUE;
11        set_state( 1 );
12       break;
13    case 1:
14       INT_LED_OFF_GREEN;
15       INT_LED_ON_RED;
16       INT_LED_OFF_BLUE;
17        set_state( 2 );
18       break;
19    case 2:
20       INT_LED_OFF_GREEN;
21       INT_LED_OFF_RED;
22       INT_LED_ON_BLUE;
23
24       // Set next state
25        set_state( 0 );
26       break;
27    }
28    // Wait one second
29    wait( 1000 );
```

```
30 }
```

The test was done by writing the current state of the task to UART0.
A python script were made that measures the time between each character received. The script can be seen in code 2.13.

Listing 2.13 Python code used to measure time between received byte

```
1  #!/usr/bin/python
2  import serial
3  import time
4
5  ser = serial.Serial( port='/dev/ttyUSB0', baudrate=57600 )
6
7  t = time.time()
8  while True:
9      for char in ser.read(1):
10         print time.time() - t, ","
11         t = time.time()
12
13 ser.close()
```

The output of the script were redirected to a file. After receiving 700 bytes the standard deviation and mean was calculated using matlab. The mean is 1.0089 sec with a standard deviation of 0.0042 sec.

Part of the variance is caused by the inaccuracy of the timing on the PC running the python code. If a more accurate measure was needed, a scope could be attached to the μC's GPIO. Each time the scheduler enters the task the GPIO should be set high, and when it exists the GPIO should be set low. The scopes at SDU is capable of telling the variance of the off signal. It can be concluded that the scheduler performs well.

2.5 Hardware

2.5.1 Introduction

This secion will go in depth about the extentionboard created as an addon to the AQ M4 board. The extentionboard was developed to act as a bridge between the PC and the CAN-bus using wireless communication.
v The block schematic shown in figure 2.7 was created by the auther of the report. It was then given to Carsten Albertsen who created the schematic and did rest of the creation of the PCB.

2.5.2 Block schmeatic

Mathias: Block schmeatic:	Create schematic appendix
Mathias: Block schmeatic:	Add GPS, WIFI, PCB, PC to forkortelses liste
Mathias: Block schmeatic:	Thanks to Carsten Albertsen for PCB development

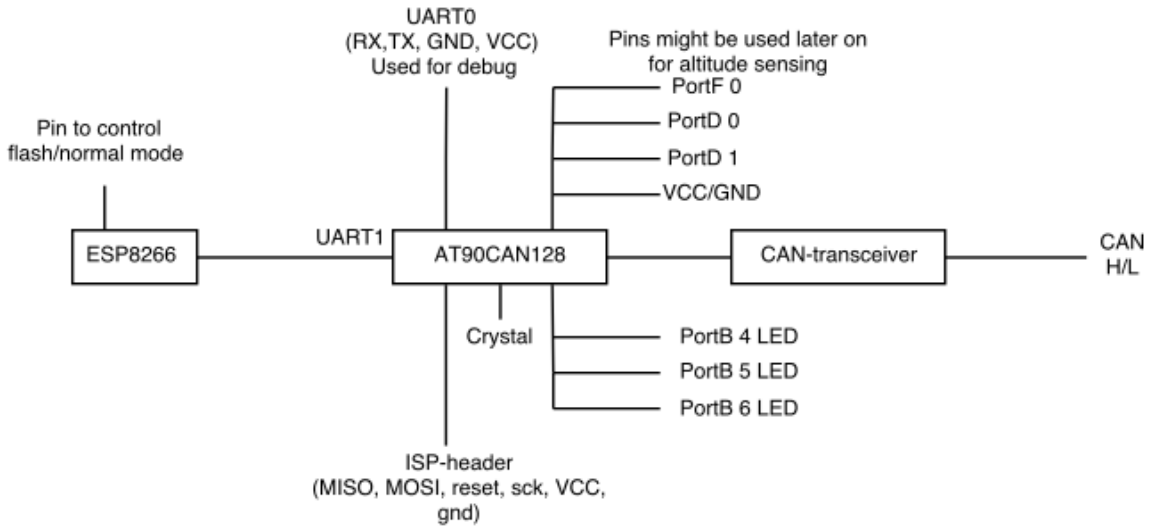

Figure 2.7 *Block schematic of the WIFI-extentionboard developed to AQ M4*

ATmega

Wireless Communication

An important part of the hardware is the wireless communication used between the PC and the drones. The wireless communication module has severel requirements it needs to fulfill in order to make the whole system work as expected. A comparison table has been made in table 2.7 to find the wireless communication module that is best suited for the task.
The following wireless modules where considered and compared.

- **ESP8266**

ESP8266 is a generel purpose 32 bit SOC with integrated WIFI 802.11 b/g/n support and buildin TCP/IP stack. It can be setup its own access point or it can connect to an existing wireless network. It runs at 80MHz and can be flashed with a custom firmware. The SOC is sold as modules with different pinouts and features such as extra flash memory [10] and different antennas. The chip has been on the marked for about two years and costs approximately 7$. It has been widely used in DIY-projects due to its low price and because it requires a minimum of network knowledge to get up and running.[11] When the SOC is shipped, it comes with a preloaded firmware which either accepts AT commands or LUA scripting depending on the version of the module. These simple programming interfaces makes it quick and easy to interface the cheap.
This leads to a large community where most of the problems have been found and solved already. Arduino has been ported to ESP8266 which makes it even easier to get it up and running. Their official Arduino GitHub has 2125 commits on their master branch at the time of writing[12]

[10] https://www.olimex.com/Products/IoT/MOD-WIFI-ESP8266/open-source-hardware
[11] http://www.esp8266.com/ - 43.000 posts in forum
[12] https://github.com/esp8266/Arduino

- **EMW3165**

EMW3165 is a SOC much like the ESP8266 supporting 802.11 b/g/n WIFI with buildin TCP/IP stack. As with ESP8266 it supports setting up an access point aswell as connecting to an existing network. It has a Cortex-M4 μC which run at 100MHz. It supports custom firmware and can be aswell be bought as different modules with different pinouts and antennas. It differentiates itself from the ESP8266 by its higher frequency its 5 volts compatible pins [13] which makes it easier to connect other hardware which run 5 volt without the need of a logic level shifter. It has been on the marked for only one year and costs approximately 9$. Since it is a newer board than ESP8266 it has not been used in the same number of applications and thereby has a smaller community behind[14]. Their most active GitHub has 147 commits on their master branch at the time of writing[15].

- **nRF51822**

nRF51822 is also a SOC, but it is using Bluetooth instead of WIFI. The nRF51822 μC is implementing BLE which is a power efficient way of sending and receiving data. The chip supports broadcasting which could be used in this project. The μC can be bought as a standalone component or mounted on modules as the two other μCs. Different modules offers different types of antenna connectors or buildin antenna on the PCB. It has not been possible to find an Arduino ported firmware that supports this μC. To write a firmware for the μC it has to be done using Nordic Semiconductor's proprietary SDK.

- **XBee**

XBee is a module that implements the Zbee standard. The Xbee modules work as a wireless serial connection. The Xbee modules supports mesh networking which means the modules by themself figure out which module is closest and makes the connection. This idea makes sense in this application since there will be multiple drones and one computer. If one drone gets too far from the PC, it can just connect to one of the other drones closer to the PC.
The Xbee solution is ready to use and requires a minimum of programming to get up and running. The modules also support GPIO for digital in and output and analog input.

Mathias: Wireless Communication: 'footmotemark til at referere til samme fodnote flere gange

Product	Size	Weight	Price	Documentation	Range	Score
ESP8266	24x16mm[16] {7}	1.5g[17] {8}	7.5$[18]	{9}		
EMW3165	32x16mm[19] {6}	5g[20] {2}	9$[21]	{3}		
nRF51822	18x10mm[22] {8}	3g[23] {0}	7.5$	{2}	30 Meters[24] {9}	
XBee	24.38x27.61mm [25] {3}	3g [26] {4}	25$[27]	{9}	91 Meters[28] {9}	

Table 2.7 *Comparisontable used to compare different wireless communication modules*

Mathias: Wireless Communication: vægt på ESP til 3 gram med link - det er hvad der er oplyst. Efterfølgende skrive at det er vejet til 1.51 gram(hen mod slutningen)

[13] https://hackadaycom.files.wordpress.com/2015/07/emw3165.pdf
[14] http://www.emw3165.com/ - 200 posts in forum
[15] https://github.com/SmartArduino/WiFiMCU

The products compared in 2.7 are chosen to have approximately same specs. Onboard antenna, breakout for easy pin access.

Pins

A few pins where made available through solder pads for easy access if needed later on.

The following pins where available as solder pads:

- PortF 0 - Alternative function as ADC, channel 0
- PortD 0 - Alternative function as interrupt, INT0
- PortD 1 - Alternative function as interrupt, INT1

In case the onboard baromter isn't accurate enough, an alternative distance could be used to measure the drones altitude with respect to the ground. PortF0 has been made available since some distance sensors give output as an analogue value. An example of such sensor is an Infrared proximity sensor.[29]
As an alternative type of distance sensor, a ultrasoinc could be used such as HCSR04. As output it gives a binary output with high-time proportional with the distance.[30]. To detect the high-time, one of PortD1/0 would be useful.

Which type of sensor suits best as a distance sensor to provice altitude information to the drone is ouf of the scope of this report. The PCB has just been made ready to different types of sensors.

Debug/ISP

In the final schematic UART0 and ISP pins where combined in one pinheader for easy access through one cable.

Mathias: Debug/ISP: Refer to image of board

To program the AtMega the ISP pins where required to be easy accessable. UART0 was made accessable to be used as debug and programming of the ESP8266 board. The plan was to setup the AtMega as UART passthrough from UART0 to UART1. Due to a mistake[31] in the final schematic, both UART0 and UART1 where made accessable trough the ISP/debug header. This ended up making it easier to program the ESP8266-board without using the AtMega as UART passthrough.

ESP noteS: - GPIO15 = bootmode, low for ... - GPIO0 = flash mode
https://zoetrope.io/tech-blog/esp8266-bootloader-modes-and-gpio-state-startup

[29] http://www.sharpsma.com/webfm_send/1208
[30] http://www.micropik.com/PDF/HCSR04.pdf
[31] The wrong pair of MISO/MOSI pins where made available in the ISP-header. The correct pair of MISO/MOSI is also RXD0/TXD0 as alternative function

Chapter 3

Results

Chapter 4

Discussion

Chapter 5

Conclusion

List of Figures

List of Tables

Appendices

Bibliography

[1] Some guy. 2993.

5.1 Test of spoofed GNSS

Testing of the spoofed GNS was split intro two subtests.
The first test was conducted in order to validate the GPS spoofing using the CAN-bus works when the drone is laying on a table.
Test 1 can be seen in 5.1.1.
The second test was conducted as the first test, but when the drone is airborn.
Test 2 can be seen in 5.1.2

Figure Y shows the waypoint list uploaded to the drone. It is expected that the drone will behave as if it was using its onboard GPS. The GPS positions will be gathered in a rosbag to be used later on in test2. A flight with the onboard GPS will be done in order to have a reference flight.
Testx Test two will be conducted much the same way as test1. However the GPS will be replaced with a lightweight RTK GPS. The RTK GPS positions will also be saved in a rosbag for later analyse. Is is expected that when using the RTK GPS the drone is closer to its waypoints shown in figure Y, than when using a normal GPS.

5.1.1 Test of spoofed positions indoore

Introduction The purpose of this test was to see if the GPS coordinates was spoofed correctly and read probably by AQ and to validate the ROS node responsible for creating CAN messages was working as long as the drone is laying on a table. If this works, it shows that the UKF works with the spoofed coordinates as expected, and that the author of the report can spoof the GPS position from a vision based localization system.

A rosbag was used to collect data from the GPS since it publishes data to a topic when played as a node did when the recording took place. This results in no code has to be written and when the code works with a rosbag it also works when it is replaced with a node providing real-world data.

Test

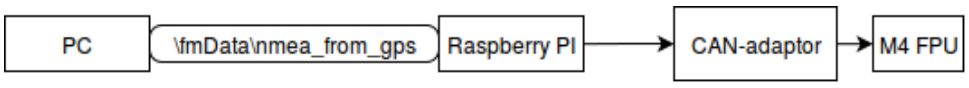

Figure 5.1 *Block schematic of the connected components*

Figure5.1 shows the connected componenets.

To get the NMEA strings from the GPS, an already existing Frobomind component will be used. It reads from a serial port, and publishes the parsed NMEA string to */fmData/nmea_from_gps*. Rosbag was then used to save the messages. The code written by the author will then subscribe to */fmData/nmea_from_gps* and publish the CAN-messages to another topic. A third node which is also a part of Frobomind will then subscribe and send the CAN-messages.

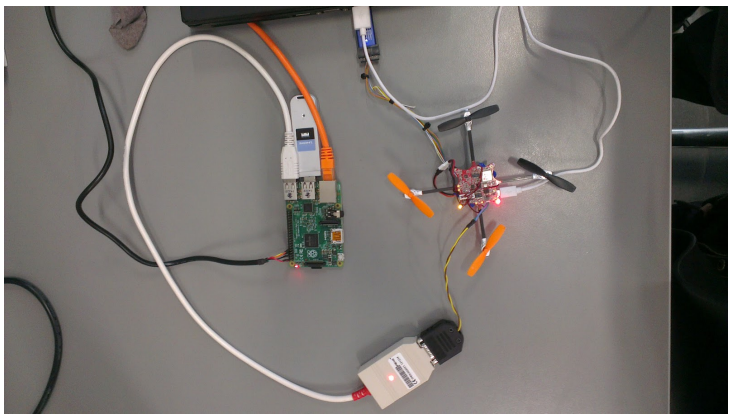

Figure 5.2 *Test-setup shown*

The Rpi is connected directly to the PC using the orange ethernet cable. The rosbag was stored and played on the PC, so the ethernet cable were used to share rostopics between the PI and the PC. The roscore where running on the PC.
The Ladybird drone is connected to the RPI using a PEAK CAN-adapter to transfer CAN messages. The Ladybird is powered by the white USB-cable and to show the position of the drone in QgroundStation. The PC is connected to the drone using ST-Link SWD to easier start, stop and restart the drones firmware during the test. To access a TTY on the PI, an FTDI cable has been connected to the PC.

In order to obtain GPS-testdata to spoof into AQ, a rosbag was used to record data collected by the GPS mounted on the authors laptop. The setup can be seen in figure 5.3

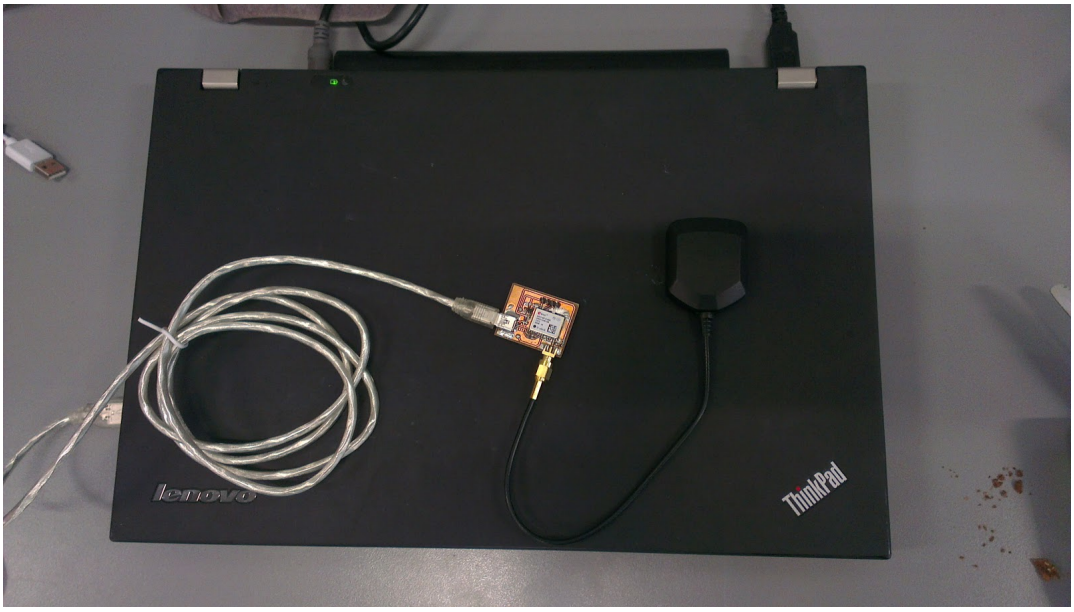

Figure 5.3 *The Neo6-P GPS (PCB developed by SDU), u-blox active GPS antenna* [1] *and authers laptop used in this test*

The coordinates recorded by the rosbag is shown in figure 5.4. FreeNMEA[2] was used to plot the GPGGA messages obtained from the GPS.

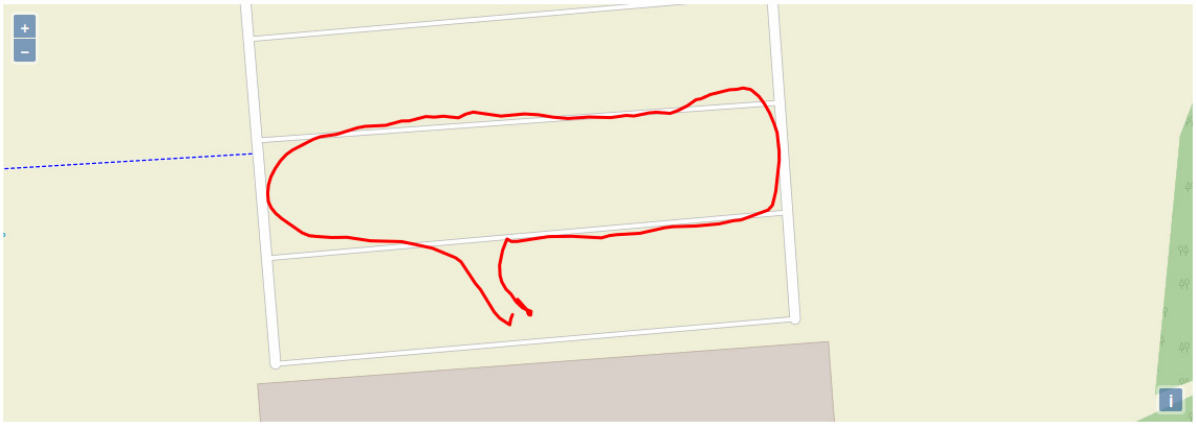

Figure 5.4 *Test coordinates plotted*

The rosbag containing the GPS coordinates where played on the Rpi, to see if the ROS node responsible for converting GPGGA messages into CAN messages were working. QgroundStation plots AQ's belief in where the drone is. The plot can be seen in figure 5.5

[2]http://freenmea.net/

Figure 5.5 *QgroundStations plot of AQ's belief in where the drone is*

Conclusion This test visually confirms that the coordinates is spoofed correctly and read probably by AQ as long as the drone is laying on a table. Passing this test means there is reason to believe that it also works when the drone is airborne.

5.1.2 Test of spoofed positions outdoore

Introduction The purpose of this test was to see if the GPS spoofing worked when the drone was airborn. The same hardware and software were used as in section 5.1.1, though with a few changes.

The GPS used in test 1 to collect coordinates were mounted on the EduQuad.

The idea was to pass-through the GPS coordinates from the Neo 6-P GPS to AQ. If the CAN spoofing works as expected the drone should behave normally when flying in position-hold mode.

Test

Figure 5.6 shows how the hardware were mounted on the EduQuad. It was temporary mounted with strips and tape to avoid short circuiting.

Mathias: Test of spoofed positions outdoore: Opdater billede med test-equipment

In test 1 the altitude of the drone was hardcoded since it had no relevance in the test. Though in test 2 if was necessary since the UKF[3] uses the altitude from the GPS to estimate the position of the drone. Furthermore the following check where inserted to make sure only valid GPS coordinates were fed into the AQ:

[3] https://github.com/bn999/autoquad/blob/master/onboard/run.c#L111

Listing 5.1 Quality checks added to discard bad positions

```
// default low accuracy
gpsData.hAcc = 5; // Meters
gpsData.vAcc = 5; // Meters

if(gpsData.fix == 1){ // Single point solution
   if(gpsData.hdop < 3){
      if(gpsData.satellites >= 5){
         AQ_PRINTF("Recv. Valid GPGGA\n",0);
         // High accuracy
         gpsData.hAcc = 2.5; // Meters
         gpsData.vAcc = 2.5; // Meters
         // Set flag to process new message
         CoSetFlag(gpsData.gpsPosFromCanFlag);
        } else {
         AQ_PRINTF("Satellites: %f \n", gpsData.satellites);
        }
    } else {
       AQ_PRINTF("HDOP: %f\n", gpsData.hdop);
    }
} else {
   AQ_PRINTF("Fix: %f\n", gpsData.fix);
}
```

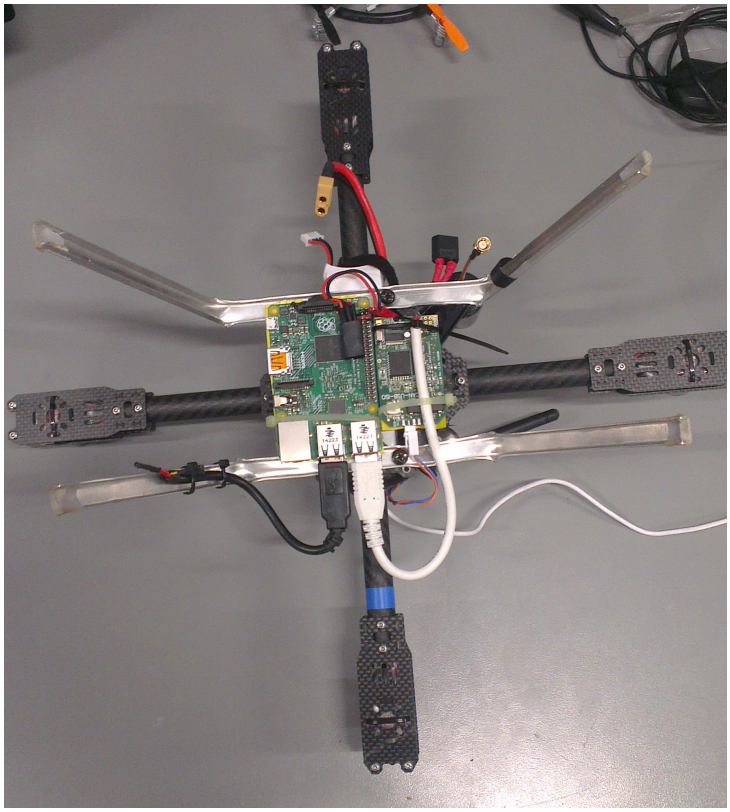

Figure 5.6 *The EduQuad drone with the hardware used in the test*

An extra battery was mounted on the drone to supply the Rpi. The ROS nodes running on the Rpi had to be start up before AQ starts registering notes on the CAN-bus. If the nodes was not started up, they would not reply to AQs reset-msg it sends when it is booting.

When the ROS-node were connected to the CAN bus, it showed that the ROS-node needed extra checking on CAN messages in order to detect difference between messages targeted the ROS-node and messages targeted the ESC32-nodes.

A bug in the *can_socketcan* -node used to communicate with the Peak-Can adapter where found a fixed. It was later on merged back into Frobomind [4] **Conclusion**

The test did not go as expected. The test was carried out under a time pressure since the author's supervisor only had 30 minutes to carry out the test.

The author did though gain experience when having multiple CAN devices connected to the CAN-bus.

Several things went wrong through the test:

- The onboard GPS was not disabled
- The author accidentally forgot to set the DOP values used in AQ
- No reference flight after the firmware were flashed for the first time.
- The GPS antenna was not mounted in the middle of the frame as the onboard antenna.

Figure 5.7 shows the DOP value is lowering witch indicates that AQ was using its onboard GPS.

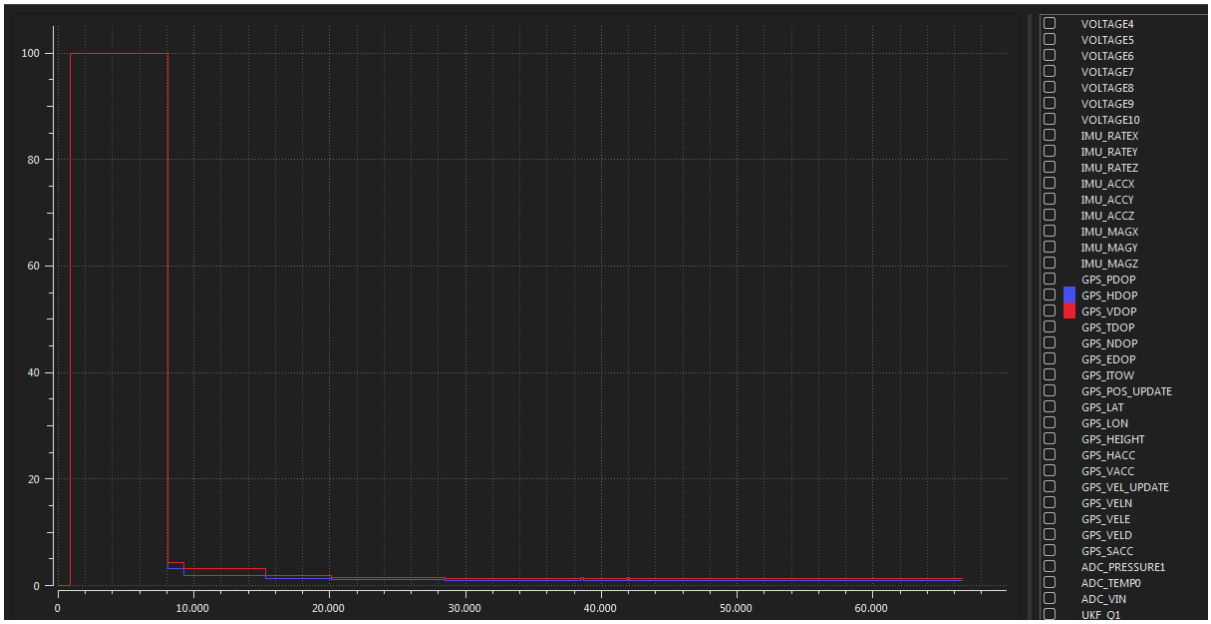

Figure 5.7 *Plot of h/vDOP from logs when the drone was airborn*

The test will be carried out again.

[4] https://github.com/FroboLab/frobomind/pull/12

5.2 Documentation made through out this thesis

Mathias: Documentation made through out this thesis: Ja, men gør det som elektronisk bilag (pdf) og beskriv i rapporten, at den ligger der. Marker det evt. dine bidrag med farve, så det er meget let at se