Group 11 | Jetse Brouwer 4615964
CS4140ES | Niels Hokke 4610148
Embedded Systems Lab | David Enthoven 4502124
June 26, 2018 | Nilay Sheth 4737490

Abstract

In this report, we present our implementation and results for the quad-copter assignment. We explore the possibilities of running FreeRTOS on the nRF51822 ARM Cortex M0, which gives us flexibility in terms of task priorities and scheduling. We developed a more interactive GUI using python to replace the PC-terminal, helping us visualize the modes, throttles and gyro values from the drone. Using interrupt driven joystick handling and FreeRTOS queues, we achieved quick serial communication. full-control mode using the on-board DMP was successfully implemented. Finally, instead of the DMP, we designed our own filters. Using plots we show we can generate a trustworthy signal without the DMP.

Introduction

The TU Delft course CS4140ES Embedded Systems Lab requires its participating students to complete a practical assignment. The goal of these assignments is to broaden the student's practical knowledge on know how to implement time-critical programming and control systems on an existing hardware platform. This assignment specifically focuses on programming a QuadCopter in C. One of the biggest challenges is using a processor that has just enough processing power to perform all the desired functions. These functions include communication with a PC 'ground station', maintaining equilibrium, checking on board sensors (barometer, gyro, accelerometer and battery voltage), closing the feedback loop to power the motors and finally responding to emergency situations. This document describes the solution to the stated problem as designed by Group 11. First, the architecture of the system will be described, then various implementation details will be presented such as the FreeRTOS and the GUI. Having made the basic software work on both ends, the next section will explain about the Control Loops. With these implementations some experimental results will be listed. Lastly, the key takeaways will be described in the conclusion chapter along with some discussion.

1 Architecture

In this chapter the structure and architecture of the multiple hardware and software elements are discussed.

1.1 Hardware architecture and interfaces

A major part of keeping the drone afloat is interfacing with the supplied hardware. There are several components which need to be controlled to maintain a fully functioning drone. At first the computer must be able to read out the joystick position and combine this data with keyboard input. The communication between the PC and the drone are realized using UART.
Besides that the drone is equipped with a module called "GY86" which has a gyroscope, accelerometer, magnetometer and barometer on board which can be used to determine the attitude and height of the drone. Communication with this module is done using I2C. For controlling the motors, the nRF communicates to the ESCs (electronic speed controllers) using PWM/Timer signals. For safety reasons the drone should also periodically check the battery voltage and detected dangerously low voltages, this is done using the on-board analog-to-digital converters. For further convenience to the user the drone is supplied with a number of LEDs which can be used to inform the user about the states and modes the drone is in (further explained in Section 2.3: UI).

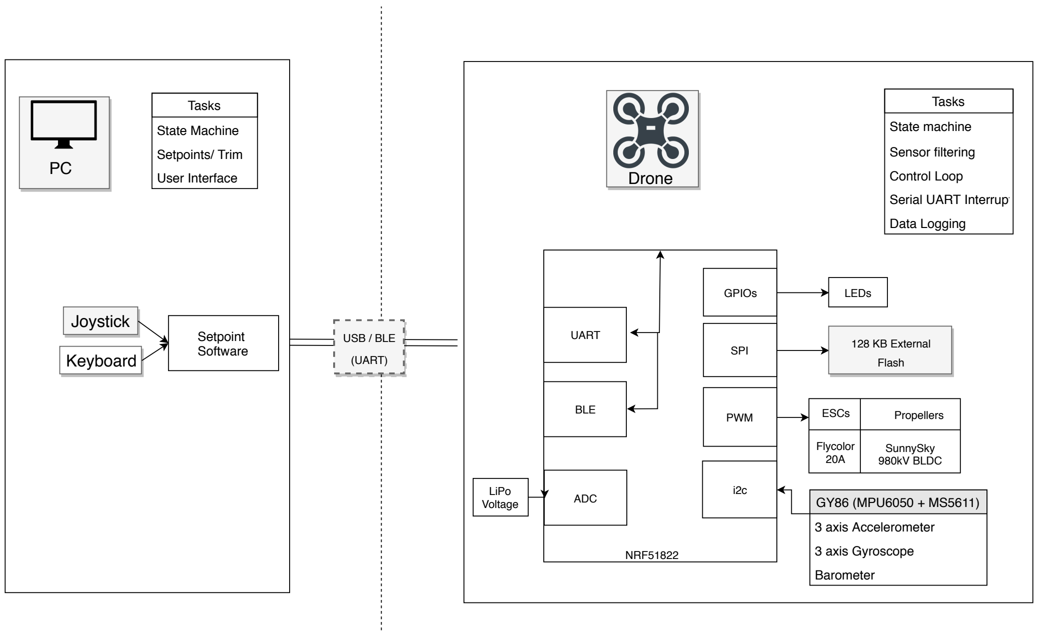


Figure 1: Overall Architecture

1.2 Software architecture

The embedded software is implemented using FreeRTOS. This offers us task preemption by offering thread scheduling and stable response times, even after code modifications. This enables us to flexibly add auxiliary functions with a lower priority running concurrently with the control loop without interference of other higher priority tasks.
This does come at the price of a higher memory footprint and a (close to) negligible run-time overhead. The run-time overhead is however easily compensated by the fact that when giving the control loop the highest priority, adding any tasks will not result in higher response times (given that the task set remains schedulable). The tasks and the accompanied timings are displayed in Table 1.

Table 1: The priorities and period of the tasks in FreeRTOS

Task	Priority	period
check_Battery_voltage	1	$10ms(10Hz)$
sensor_loop	1	1000 Hz
validate_ctrl_msg	2	(interrupt based)
validate_para_msg	2	(interrupt based)
Control loop	3	$9ms(\pm 111.11...Hz)$

The control loop is the task responsible for reading out the sensor (when in DMP mode) and calling mode specific functionality. It has been given the highest priority as we deemed it logical that keeping the drone stable is the main priority. The frequency has been chosen so that the control loop executes slightly more often than the GY-86. If both were to be set to 100hz the GY-86 would generate data slightly faster than the control loop reads out the data, resulting in the buffer slowly filling up with samples. The 'validate_ctrl_msg' and 'validate_para_msg' are used to verify and effectuate the data sent along with the messages. The UART interrupt service routine (ISR) only receives the bytes coming in, and then passes them on to one of these tasks for further handling. The handling of the messages is separated based on the prefix of the message. The sensor_loop is only used for reading the raw data from the GY-86 and running the Kalman filter. Unfortunately, we did not manage to test the raw mode feature on the drone within the deadline. The check_Battery_voltage task is responsible for checking the battery and reporting telemetry data back to the PC. The telemetry is send roughly every 100 ms, however the battery is only checked once every ten runs, effectively checking the voltage every second.

2 Implementation

This chapter discusses how the control loops, serial communication, and user interface where implemented.

2.1 Control Loops

For both DMP and raw mode the same control loop is used, as raw mode is designed to overwrite the same global variables as the DMP mode would (requiring only re-tuning of the drone). Because yaw mode is also part of full-control mode they are implemented as one function, with a parameter (Boolean) to enable or disable the pitch and roll control. The **Yaw mode** is implemented as follows:

1. Adhering to convention, the twisting of the joystick handle assigns the drone with a particular set-point yaw rate. In all the other axes, the joystick handle assigns the position/attitude of the drone.
2. This is very useful for all that has to be done now is to compare the set-point yaw rate (set with the joystick) with the yaw rate coming from the gyroscope.
3. If the twisting of joystick were to set the position of yaw instead of the yaw rate, then it would require us to implement something like the cascaded P controller as mentioned ahead in the assignment description. However, since the accelerometer is useless in this "yawing" about z axis (ref Figure 5), it would be impossible to implement this cascaded controller. Magnetometer to the rescue, the DMP fuses the magnetometer readings to extract the values of the absolute "yaw" about z-axis (wrt to calibrated North). It is labelled as the variable 'psi' in our case.
4. The yaw mode is implemented using the 'sr' variable (The angular velocity around the z-axis) as a simple P-controller. The set-point is first scaled to roughly match the order of magnitude of the gyro output. It then is multiplied by a settable P-value for tuning.

```
yaw_output = ((- SetPoint.yaw << 6) - sr) * (P_YAW);
```

Listing 1: Yaw control

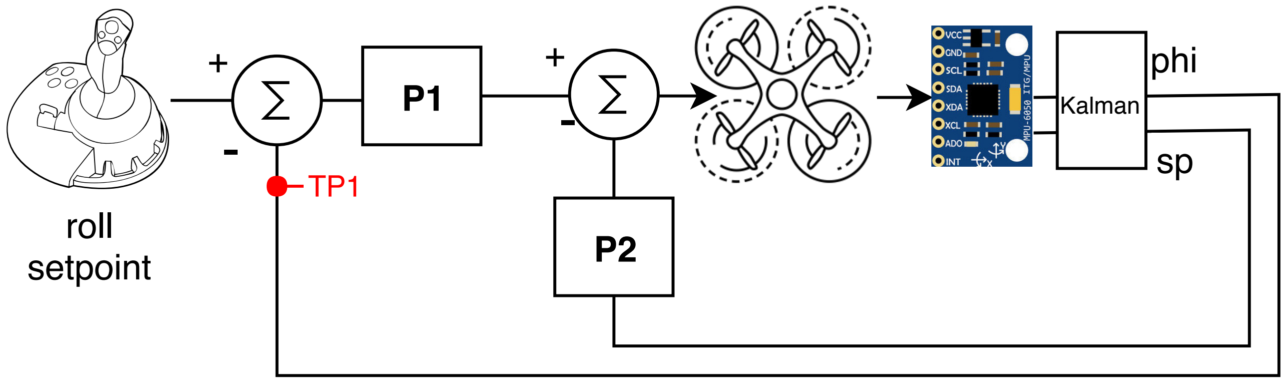


Figure 2: Cascaded P control using Kalman filters

Now we move ahead to discuss the **Full Control mode**, in which the attitude of the drone about all the 3 axes is completely controllable.

1. The normal intuition to close the loop would be to integrate the moments twice, to obtain angles and compare it with the joystick set-points. The error could be feedback into a P controller, which would then ensure adequate enough input to stabilize the drone.
2. However, the moment we say cascaded double integrators, we immediately become unstable, since each integrator fundamentally adds a phase difference of 90°. The combined 180° phase shift would result in positive feedback and saturating behaviours on the propellers of the drone. We hence switch to the concept of cascaded P controllers taught in class.
3. After writing down the expression for the cascaded P controller, it is seen that it strongly resembles PD controllers used in Control Design. Listing 2 is the PD controller implemented to make the drone hover about its neutral point. Gain $P1$ is the proportional gain to the controller which decides the aggressiveness of the drone to reach a neutral set-point angle. Gain $P2$ is the

derivative gain which decides the reaction of the drone to change in angles. As a result, the gain $P2$ was tuned first (by giving $P1 = 0$), by checking its reaction to change in angles.

```
roll_output  =    P1*((SetPoint.roll *48)-(phi << 1))  -P2*sp;
pitch_output = -1*P1*((SetPoint.pitch*48)-(theta << 1))-P2*sq;
```

Listing 2: PD control equation

4. In the above formula the number 48, and the shift to left are remainders from earlier attempts on calculating an exact value, these number were later on changed multiple times. To combine the outputs of the controller we simply sum them (in 12 bits fixed point integer format).
5. The poll and pitch compensations are fed to the motors using the following lines:

```
tempMotor[0] = (SetPoint.lift  << 13) + yaw_output - pitch_output;
tempMotor[1] = (SetPoint.lift  << 13) - yaw_output - roll_output;
tempMotor[2] = (SetPoint.lift  << 13) + yaw_output + pitch_output;
tempMotor[3] = (SetPoint.lift  << 13) - yaw_output + roll_output;
```

Listing 3: Pitch and Roll control

6. Here the lift is shifted with 13 instead of 12, equaling a multiplication by 2. This maps the maximum lift to 512 instead of 1000, but it is proven that this is enough to fly the drone. As the lift is quadratic with the RPM, the lift has a square-root curve to it, however this is done at the PC side.
7. After this, the values are shifted to the right by 12, and are checked if they lie within the specified range of 0 to 750.
8. All variables used are 16 bits (or smaller) and all calculations are done in 32-bit signed, so no overflows should be expected, however since everything is signed checking for overflows easy (as all values smaller then 0 are limited to 0).

2.2 Serial protocol

To control the drone from the PC, a communication protocol is needed to transfer information. We decided to use two types of messages: Control type messages which contain the joystick set-point data, and Parameter type messages which can be used to change any parameter on the drone.

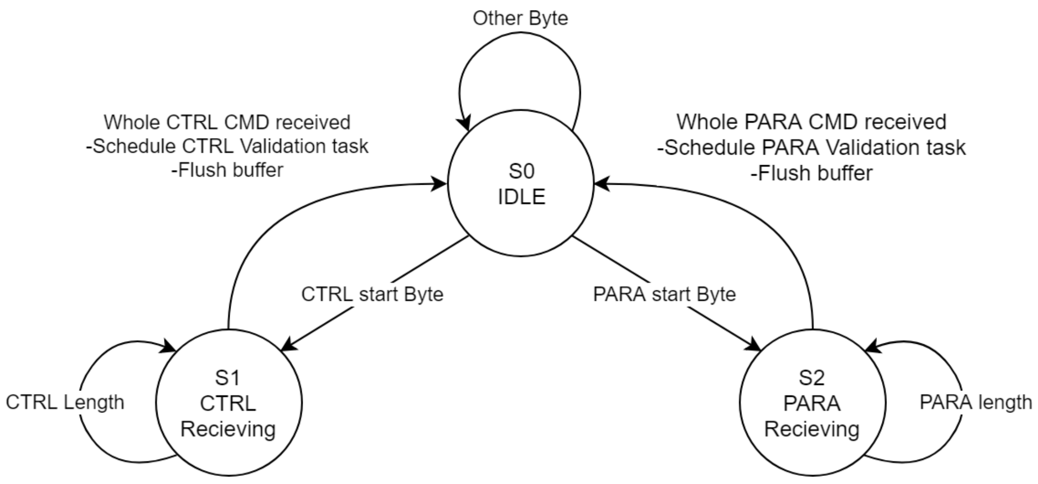


Figure 3: Serial State-diagram

Figure 3 shows the serial state-diagram on the drone side. The system starts in the IDLE state (S0) and waits till a CTRL or PARA start-byte is received. When one of the two start-bytes is received it waits till the whole message is received (S1 & S2), and schedules a FreeRTOS message-validation task. The whole serial state-diagram runs in the UART-ISR. FreeRTOS queues are used to store the messages before the validation tasks can run. There are two validation tasks, one for each kind of message. The tasks are in a yielding state until there is a message in the queue. Once a message arrives and no higher priority task is running the validation tasks check if the CRC is correct and then parse the message.

Every time any byte is received a FreeRTOS xTimer is reset, making sure that when no bytes are received for more than one second the drone assumes the serial connection is broken. When the xTimer runs out a serial-timeout-function is called which sets the drone into PANIC mode.
Table 2 and Table 3 show the two types of messages. Both messages have the same structure of start-byte, data-bytes, followed by a CRC byte. The CRC is calculated over both the start-byte and the data-bytes. The data-bytes in the Control message exist of Yaw, Pitch, Roll, and lift bytes, where one byte is used for each data-field. These values come almost directly from the joystick, only a optional trim is added before the data leaves the PC.

Table 2: Control message

Start-Byte	Yaw	Pitch	Roll	Lift	CRC
0xAA	1 Byte	1 Byte	1 Byte	1 Byte	1 Byte

Table 3: Parameter message

Start-Byte	Reg.address	Data (msb)	Data	Data	Data (lsb)	CRC
0x55	1 Byte	1 Byte	1 Byte	1 Byte	1 Byte	1 Byte

The data-bytes in the Parameter message exits of one Register-address byte and four data bytes. The Register-address byte can be used to indicate to which place in the register map to write. Table 4 shows the layout of the register map, some variables exist of multiple bytes indicated with a (h) High and (l) Low byte. The register-map is an uint8_t array which is updated every-time a parameter message is parsed in the parameter-message-validation task.

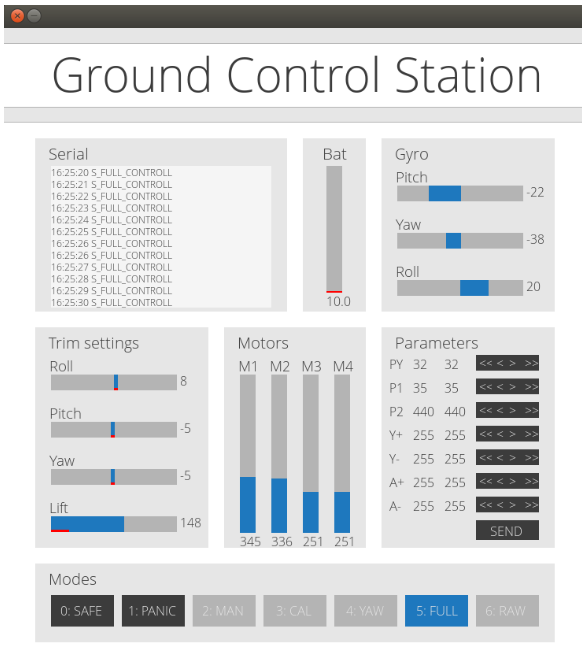
Table 4: Register map

Address	Data (msb)	Data	Data	Data (lsb)	Description
00	-	-	MIN_LIFT	MAX_RPM	Bounds
04	-	-	-	MODE	Mode
08	-	-	P_YAW (h)	P_YAW (l)	Yaw control
12	ANGLE_MAX	ANGLE_MIN	YAW_MAX	YAW_MIN	Bounds
16	P1 (h)	P1 (l)	P2 (h)	P2 (l)	P1, P2

On the PC side 'pygame' is used to read out the joystick. When the joystick is moved an event is triggered which reads out the new joystick orientation and immediately sends a control message. The joystick updates at a max rate of once every 15ms. If the joystick is not moved and no event is triggered we send a control message every 400ms as a keep alive message.
The data being send from the drone to the PC is simply streamed. As this information is only used for live displaying and debugging, it's no big problem if some messages go missing.
Using setserial and the command "setserial [Serial_Port] low_latency" the serial kernel of the OS is put into low latency mode which brings the round-trip time to 7-9ms if no higher priority tasks are running. With the higher priority control task running the serial round-trip time goes up to 13-15ms.

2.3 User interface

The LED's on the ES were used to indicate certain states of the drone. When the ES is in save mode the green LED is on and when in panic mode the red LED is on. Also an continuous blinking blue LED indicates the device is running and has not crashed. An blinking red LED indicated an stack-overflow has occurred and the program is no longer running.
In order to provide the pilot with more direct feedback about the current state of the drone, a interactive graphic user interface was designed. Early on in the project it was decided to implement the GUI in python making use of the pygame python package. This package has build in features to handle joystick and keyboard and to draw basic shapes on screen. This joystick handling feature was especially handy since it handles new incoming joystick messages as a sort of interrupt event. This meant that no polling was required by the PC and also a very responsive joystick implementation was developed. A scaled down version of the GUI can be seen in Figure 4.



```
16:25:20 S_FULL_CONTROLL
16:25:21 S_FULL_CONTROLL
16:25:22 S_FULL_CONTROLL
16:25:23 S_FULL_CONTROLL
16:25:24 S_FULL_CONTROLL
16:25:25 S_FULL_CONTROLL
16:25:26 S_FULL_CONTROLL
16:25:26 S_FULL_CONTROLL
16:25:27 S_FULL_CONTROLL
16:25:28 S_FULL_CONTROLL
16:25:29 S_FULL_CONTROLL
16:25:30 S_FULL_CONTROLL
```

PY	32	32
P1	35	35
P2	440	440
Y+	255	255
Y-	255	255
A+	255	255
A-	255	255

Figure 4: The GUI

In addition to the handling of the keyboard and joystick inputs the ground control station also provides active visual feedback about the current state of the motors, battery and the gyroscope, the latter being especially useful for calibration. Furthermore a live feed of specific serial messages is implemented which aided with debugging. The implemented states were made available as color-coded clickable buttons, blue indicating current state and light grey indicating impossible states. In order to quickly tune the parameters a bunch of clickable buttons was implemented witch increased or decreased certain parameters in different step sizes.

3 Filtering

Without the DMP, the raw sensor values as susceptible to all kinds of noise, and cannot be trusted straightaway. For full control of the drone, it is essential that we have good values on ϕ, θ, ψ. For understanding filtering in detail, we first look at the axes of MPU (GY-86). Figure 5 explains the variable names associated with the 6-axes that are crucial when filtering to obtain the attitude of the drone. The axes in red indicate the accelerometer axes and axes in black indicate the gyroscope axes. It is quite clear that rotation about one of the axis will change the readings across some other axes as well. For instance, if there is a rotation about the *sp* axis, there is a change in reading on the *sax* and *saz* axes, since now a different component of gravity acts along these axes. Before rotation, the entire component of gravity was along the *saz* axis, and the component of gravity along *sax* was zero. Figure 6 explains the axes readings associated with the Roll and Pitch movements of the drone, which are filtered using a Kalman filter.

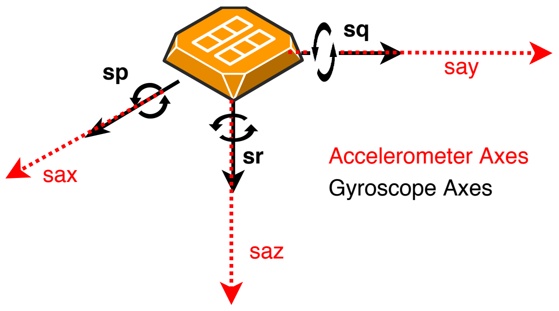


Figure 5: MPU 6050 axes - labelled

3.1 Kalman filters

In this section we discuss why using Kalman filters is a good choice on filtering the Pitch and Roll angles of the drone. Normally, the accelerometer readings are found to be quite noisy (red line in Figure 7), and even pick up vibrations easily. The source of noise is suspected to be the high sampling frequencies and the vibrations on the frame via the motors of the drone. Although, this noise is quite well distributed about the mean without any bias. We exploit these unbiased readings, while discussing

Kalman filters. As seen in Figure 6, while pitching there is a change in the acceleration experienced by accelerometer's *sax* axis while the drone moves with a particular velocity changing the readings on gyroscope's *sq* axis. The gyroscope is known to be less affected by vibrations, and is always trusted when estimating the Yaw/Pitch/Roll velocities. However, we are also interested in estimating the Pitch and Roll angles as required by the PD control in Listing 2. The Kalman filter is expected to give noiseless and unbiased readings of the angles given the raw angles and velocities of rotation. To achieve this, we use Listing 4 from [1].

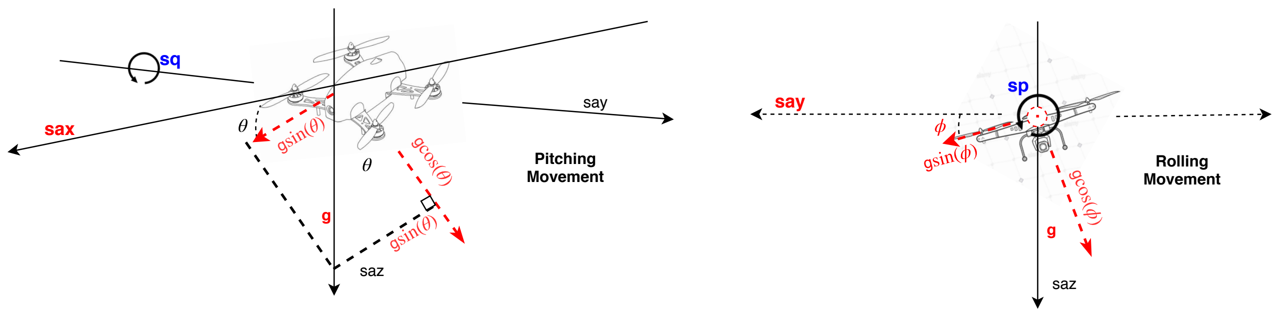


Figure 6: Left: changes in axes while pitching, Right: changes in axes while rolling

```
#define MUL_SCALED1(a,b,scale) ((a*b) >> scale)
// C1 = 3; C2 = 10;
p      = sp     - p_bias;                       // remove the bias
phi_f  = phi_f  + MUL_SCALED1(p, p2phi , 14); // integral vel = angle
phi_f  = phi_f  - ((phi_f - say) >> C1);       // sensor fusion
p_bias = p_bias + ((phi_f - say) >> C2);       // update rate of bias
```

Listing 4: Kalman filter

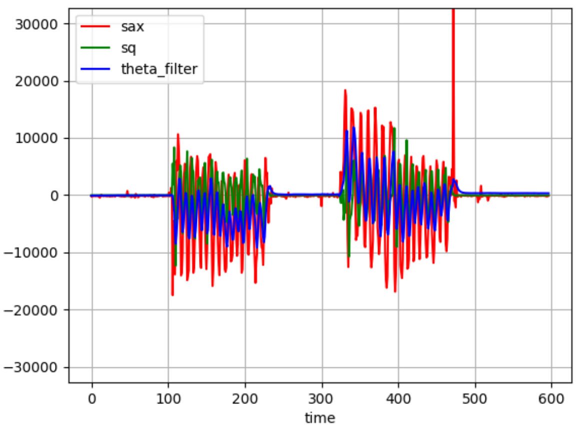


Figure 7: Effect of slow bias updates

The filter implemented uses fixed point arithmetic to do quicker calculations on the nRF51822 chip. While implementing the filter some observations were made by varying the $C1$ and $C2$ gains. With high values of $C2$, there was an inherent bias (refer blue line in Figure 7), as if the integrator could not clear its accumulated values. Even when the board was brought back to a neutral position, there was a steady state error in the angles given out by the filter. So the bias was adjusted first to give quick responses and zero steady state errors. Later the signal was checked for its response to jitters. Initially the filter jumped along with the accelerometer to give spikes in the angle's estimate. The filter was then made to trust the gyroscope's reading more by varying the $C1$ gain.

3.2 Butterworth filter

In this section, we explain why using Butterworth filters are used for extracting the yaw velocity. In the above section, we could apply the Kalman filter for pitch and roll angle estimation since the gyroscope had an axis to check against the accelerometer as can be seen in Figure 5. A noise-less sensor fusion was hence possible. Now if we carefully observe the axes of the the MPU, in case of yaw moments we find that the accelerometer has no change in reading for such movements in the yaw axis. When we give a yaw moment to the drone, the components of gravity affect the accelerometer's axes in the same way as before. Hence we cannot perform sensor fusion in the way we did earlier, and must switch to another filtering technique taught in class. The second order Butterworth filters (low pass) exactly do that. They help us to filter the yaw velocity coming from the gyroscope, by only passing the frequencies relevant for neutral control of the drone. The cut-off frequency for yaw velocity readings was selected to be 10Hz, since everything above it is considered as noise during pose estimation.

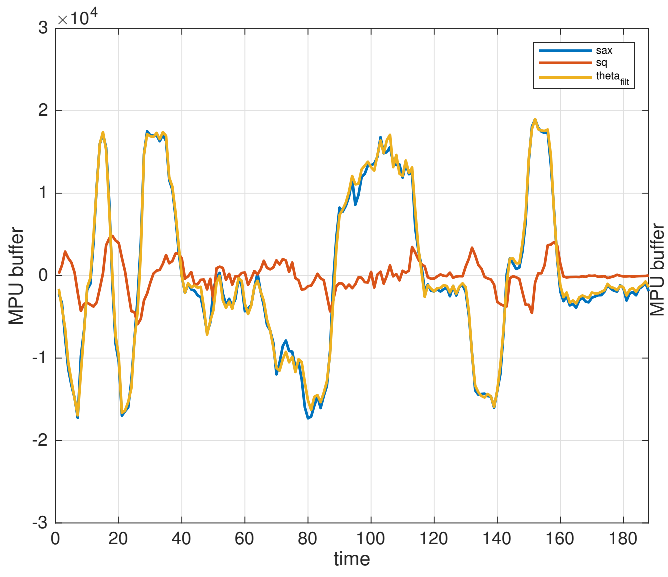


Figure 8: Pitch filter plot at 500Hz

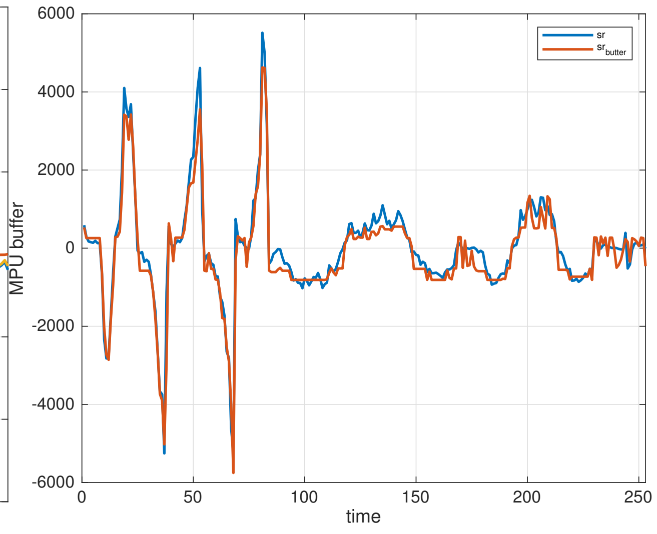


Figure 9: Yaw filter plot at 500Hz

3.3 Fixed point arithmetic

With the update rates mentioned in Table 1, we observe that not using fixed point arithmetic could create a bottleneck if the calculations have to run without a FPU. The nrf5812 can technically handle floating points without an FPU, does not have convincing performance in terms of FLOPS. This could potentially induce instability to the drone by providing corrections too late. These late corrections could also in a worst case scenario be out of phase, which would result in positive feedback instead of negative feedback. The effects of positive feedback in control systems have been known to be rather unstable. TP1 (test point 1) in Figure 2 tries to indicate the location of the phase shift in the block diagram. If the Kalman filter uses floating point arithmetic, it could lead to huge phase differences, and phase differences greater than 180° could make the sign of the summing terminal positive. This positive feedback could make the drone unstable. Hence the need of fixed point arithmetic. For the 2^{nd} order BLP filter, while rounding up numbers, it was seen that 14 bits are necessary to represent the decimal fraction in order to guarantee maximum 0.1% change in filter poles of the transfer functions as recommended in [2].

4 Altitude control

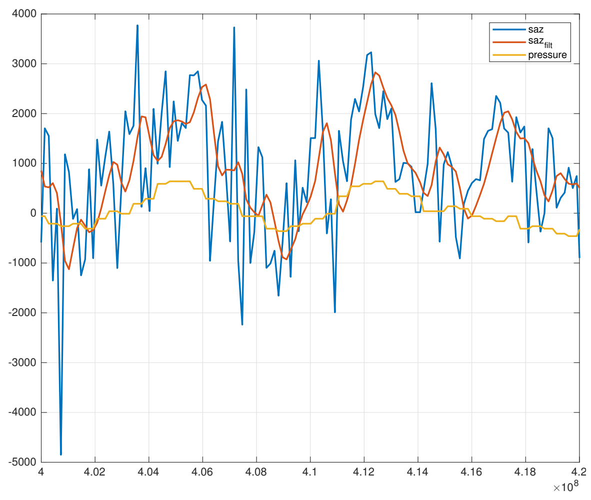


Figure 10: Altitude Butterworth filter

Altitude control is an important feature on the drone which could be used for applications like wind turbine inspection. Although it was found that the pressure sensor could give faulty readings in presence of wind. Hence, scaling the propeller's thrust simply on the basis of the altitude sensor would be inaccurate. A moving average with a reasonable window size (100 samples) could help for better estimates. But still, we need an additional sensor for sensor fusion which gives us a lower noise covariance like the Kalman filter. Time of Flight (ToF) sensors like the laser sensor or a ultrasonic could give us an additional say in the distance from the ground. However, if we recall from the above sections, we have a spare axis on our accelerometer sensor. The *saz* axis has never been used before, and can be used to calculate the gravity acting on the drone. When the drone is stationary, *saz* is positive, in case of free-fall, the *saz* would report a close to zero reading. While ascent, accelerations would rise above $9.8m/s^2$. This could be related to us as the force experienced by us when using an elevator. However, the *saz* cannot be directly used, since accelerometer is noisy and also that *saz* axis is non zero when stationary. A double integration to extract position from acceleration would simply lead to overflows. Hence the *saz* was first subtracted from its mean to have an almost zero mean. *saz* then passes through a Butterworth filter as given in Figure 10 (Yellow line is the filtered value). The deviations were now in and around zero. After closing the loop between the filtered *saz* and motor lift, every-time a push/disturbance is detected in the *saz* axis, the drone would react immediately by varying its thrust on the ground (not tested).

5 Experimental Results

This chapter contains the measured results of our efforts. We'll discuss which modes where successfully implemented, the timings of the modes, and the latency of he serial communication.

5.1 Fully functional capabilities

During the examination we where able to successfully show safe-, panic-, calibration-, yaw-, and full-mode to the exterminator. In addition the responsiveness and mapping of the joystick and he various keyboard inputs were correct. The examined c code compiled to a total size of 44364 bits (5,5 kB).

5.2 Timing

To validate our design we performed some timing measurements. Because FreeRTOS was implemented the execution frequency's could be set in a straightforward way. Additionally FreeRTOS handles timing violation in a way which ensures that deadline misses will not crash the system (not that we had any but still).
The execution time of the control loop in different modes is displayed in Table 5. In this table it is shown that the execution time, no matter the mode, we always have an average of at least 3.5 ms. As stated earlier full processor utilization would amount to a execution time of 9ms therefore we can state that the control loop on utilizes the processor on average for about 40%.
We believe that a large part of this execution time is primarily induced by two mayor factors. The overhead that running FreeRTOS induces and the standard part of the control loop which will be executed no matter the current mode.

Table 5: execution time of the control loop in various modes

Mode	Execution time (5 sample average)
Safe	± 3.515 ms
Manual	± 3.527 ms
Yaw control	± 3.570 ms
Full control	± 3.579 ms
Panic	± 3.516 ms

Aside from the control loop it is useful to know what kind of latency's we may expect in the communication. For this we measured the round-trip time of a parameter message. This is measured from the moment the message is in the output buffer on the PC side until a response message is correctly interpreted on the PC side. The round-trip times are displayed in Table 6. From this we can see that the blocking that occurs by higher priority tasks (sensor loop and battery check) causes a very substantial latency increment.

Table 6: round-trip time of a parameter message

Set-up	Round-trip time
Highest priority	± 7 ms
Normal execution priority	$\pm 13 - 15$ ms

6 Conclusion

In this chapter the key points of this project are revisited as well as some of the noteworthy aspects about this project. Furthermore an informal evaluation about the project is done as well as a short description about each persons individual contribution to the project.

6.1 Key take away

The successful implementation of FreeRTOS is one of the key features of this project. With the FreeRTOS task scheduling we found that we have an average constant processor utilization of only 40% which leaves a quite a big piece of processing power untapped. The communication is handled on interrupt base with fixed message sizes for control and parameter messages which, although very responsive, doesn't cause any problems in the scheduling.
On the PC side a user-friendly interactive user interface was developed to great success. This provides the pilot with a lot of information in a clear visual way. this was especially useful for testing and debugging.
It's a shame that the implementation of raw-control and height-control where not finished in time even though a lot of time and energy has been spend on its implementation.

6.2 Discussion

As a team we regret that we where not able to finish the other two modes and are convinced that we would have been able to do so with a bit more time on our hands. There should be enough processing power to accommodate the extra time that this takes.

6.3 Evaluation

We were quite proud to get FreeRTOS running on the quad copter, even though its footprint meant we had to make some concessions. For us this meant that the Bluetooth functionality was immediately deemed an unreachable goal. A positive result of this was that implementation of the scheduling was rather straight forward and, as we noticed later on, quite a lot of processing power remained unused.
The other thing of which we are quite pleased is the GUI, for it is a thing of beauty. It provides a lot of very useful feedback during the development. We developed the GUI in python and this meant that we were able to achieve very responsive joystick control. The main reason for this is that we were able to handle the joystick on event driven interrupts.
We have added a link to our GitHub repository for any examiner to peruse.[1]

6.4 Contribution per team member

Jetse Brouwer worked in the start of the project on porting FreeRTOS and implementing it on the ES. Additionally he implemented the main control loop as well as the yaw control and the full control.

Niels Hokke designed and implemented the serial communication on the ES side as well as the PC side. Furthermore he implemented the calibration mode and did a bunch of work on making the ground control station pretty and user-friendly.

Nilay Sheth worked on implementing the Kalman and the Butterworth filters, also tried out altitude control to work in combination with the barometer. Unfortunately we were not able to demonstrate this during the examination. However, the plots are included in the report. (Git branches: PC-terminal and filter)

David Enthoven developed a large part of the user interface on the PC side. Furthermore he implemented the panic mode on the ES with the proper safety requirements.

References

[1] Arjan J.C. van Gemund. In4073 qr controller theory. *Embedded Software Lab - EEMCS, TU Delft*, 2011.

[2] Sait Izmit. Floating-point support for embedded fpga platform with 6502 soft-processor.

[1] https://github.com/NielsHokke/DroneController