Delft University of Technology
Faculty of Electrical Engineering, Mathematics and Computer Science
Software Technology Department

April 8, 2012

IN4073 Embedded Real-Time Systems Lab Report

Group J

E.M.	(Ewout)	Bongers	1188232
L.P.	(Lennart)	Klaver	1362526
X.	(Haris)	Papadopoulos	4192141
F.H.	(Filip)	Saad	1254243

Introduction

With the many advantages gained by the use of programmable hardware over the use of application-specific integrated circuits, it becomes essential for students to acquire the multidisciplinary skills related to Embedded System development.
This report describes the process and results of designing Embedded Software to control and stabilize an unmanned quad rotor. The main challenge faced during the design process was scarcity of resources (hardware and specifications). To overcome that, an extensive process of optimization and testing was necessary.
The following chapters present the system requirements, the chosen architecture and its implementation, and the conclusions reached.

1 Addressing System Requirements

The most important requirement is safety. That is, to insure that the quad rotor platform does not get damaged or cause harm to bystanders. This translates to the implementation of different modes of operation that can be switched by the user or switched automatically when certain conditions occur.
Next, a reliable communication link is required between the ground station computer and the quad router platform. Initially a wired link was used and as the project progressed a wireless link capability was added. The communication link is used to transmit user input from the ground station to the quad rotor (uplink) and sensor output and health parameters from the quad rotor to the ground station (downlink).
On the ground station side, a graphical user interface is developed to display the data received from the quad rotor in the form of a virtual attitude indicator and a data display window that show roll angle, pitch angle, yaw rate, and communication link health parameters. The same user interface allows the user to change modes, change control loop gain parameters (for tuning purposes), and control the quad rotor via a virtual joystick or a hardware joystick.
On the quad rotor side, Kalman and Butterworth filters are developed to filter the noisy sensor output. The filtered data is then used together with the user control input to make user control of the quad rotor possible. Transformation tables are used to overcome the fixed point limitation of the on-board hardware. Also watchdogs are implemented to detect abnormalities such as loss of the communication links and switch to the appropriate mode of operation.
The challenges that were faced during the design process are mainly related to the scarcity of resources, which can be summarized in the following points:

- Restricted access to sensor output and the fact that the units in which the sensor output is presented are unknown.
- Lack of floating point arithmetics (loosing or limiting precision in calculations).
- Tight timing constraints (Instability occurs when the controllers are too slow).
- Prevention of dangerous situations by implementing panic modes and different health monitoring techniques.
- Implementing communications with limited bandwidth and different medium.

2 System Architecture

For embedded systems there are multiple software architectures possible, for example:

- Round Robin (with interrupts) architecture.
- Priority architecture.
- Realtime operating system (RTOS).

Because the performance of the quad rotor is dependent on the update time of the controllers, the controllers need to update as soon as new sensor data is available. To accomplish this, the controllers are triggered by the sensor

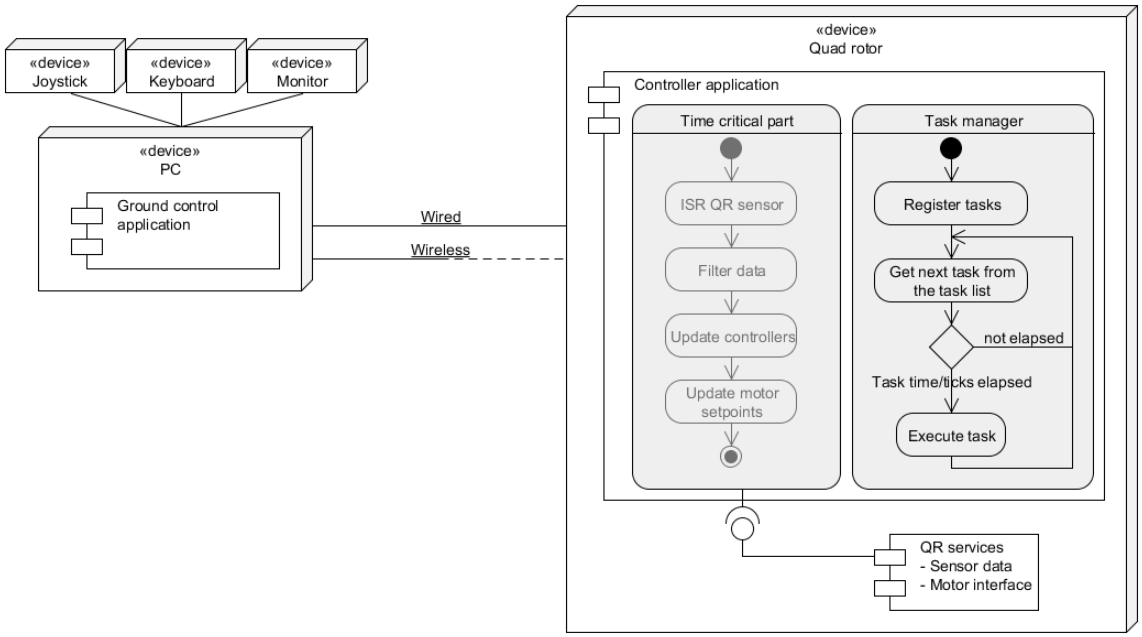

Figure 1: Global system architecture.

interrupt. This way the control loops update every time there is new data. The system has more tasks than just updating the controllers, so the system is split up into two levels:

- Time restricted tasks like sensor data filtering, controller calculations.
- General tasks like communication to pc, performance logging, internal housekeeping.

So an architecture that is real time, but also can handle lower priority tasks, is needed. The architecture used in this project is a sort of RTOS architecture. There is no OS in the sense of memory manager, but more in the sense of a task manager. The time restricted functions are triggered by the sensor ISR. All other tasks are executed by the task manager, each task with its own priority and execution time-out.
An example: to blink a led every half a second, create a function that toggles the led on or off (like led.state != led.state). Add the pointer to this function to the task manager with time-out value 500 [ms]. The task manager executes the function every half a second, and so the led blinks.

A possible trap is one easily overlooked: because the ISR has higher priority than the task manager, if the interrupt takes around the same time to execute its ISR as the time between two interrupts, then the task manager will have a hard time to execute all of its tasks regularly.

Fortunately the ISR takes half of the interrupt frequency, so there is enough time left for the task manager. In numbers: the interrupt calls every 781 microseconds and the ISR takes around 500 microseconds, so the ratio is 64% of the time for time-restricted parts vs. 36% for the task manager. These ratios are a bit biased, because in between there are small interrupts that occur sometimes (like the receive-ISR for pc communication). To get an overview of the global system architecture, see figure 1.

3 Implementation

After deciding the global architecture, the components are divided such that work could be done concurrently. The division was made based on the complexity of implementation, component priority, personal skills and preferences. This results in the following initial division of components: Ewout and Lennart created a basis for the x32 program (main, task and state components), Fillip was finding the control equations and implementing the controller component, and Harris was logging sensor data and developing filters.
After the groundwork for the x32 was done, the division of work was revised. Ewout was implementing the PC ground station, Fillip and Haris were fine tuning the the filters and controller, Lennart was implementing the x32 communication protocol and general troubleshooting.
This last would remain the division of work throughout the rest of the project. Although by no means a strict division, as the more difficult parts are mostly co-authored or at least required peer consultation. To support teamwork the well known Subversion software was used for versioning.

3.1 General Application

As discussed in architecture, the task manager controls the system tasks. The control loops and the filtering run by means of an Interrupt Service Routine (ISR), called on 1280 Hz. Above these functions runs the time testing system which can be enabled or disabled. The total system has 1960 lines of code (compiles to 59KByte).

3.1.1 Task manager

The following tasks are used:

blink_activity_led Blink a led (every 500 ms).

comm_process_message Process the messages in the incoming data buffer (every task cycle).

check_pc_link Check if the pc link has new data and thus is still working (every 300 ms).

check_qr_link Check if the QR link has new data and thus is still working (every 30 ms).

check_battery_voltage Check if the batteries have enough power (every 300 ms).

comm_send_current_state Send the current state and attitude back to the pc (every 100 ms).

The following functions are only enabled for testing purposes, not in real flight:

cpu_rotate_trapped_function Time test the next function from the function list (every second).

cpu_test_instrumentation Time an empty function to remove the test method overhead (Every 10 task cycles).

isr_qr_sensors Simulate a QR interrupt by calling the ISR (every task cycle).

3.1.2 Communications

To provide a link between PC and QR, a message system is used. This message system is designed to fit the slowest link (9600 baud) and still meet the update frequency requirements. 9600 Baud on a serial link means 960 bytes per second with a message lenght of 6 bytes, so it takes 6.25 ms to send an update message. This way an update rate of 10hz (100ms) is easily met. In combination with the sensor ISR frequency of 1280Hz this means that the control loop runs eight times before receiving a new setpoint.
The message bytes from the PC are formatted as follows: The first bit of the first byte starts with a 1, all the other bytes have a 0 as starting bit. The ordering is as follows, *1ppppppp 0rrrrrrr 0yyyyyyy 0ttttttt 0ssssmmm 0xxxxxxx* (pitch, roll, yaw, throttle, switch message type, mode, checksum per byte). To update controller gains for example, the s bits can

change the meaning of the message to contain values for different gains. This way it is possible to use the same message format for different subjects.
Incoming messages generate an interrupt on the processor. The interrupt service routine takes the byte from the register and places it inside a buffer. If too much data is received (buffer overflow) then the incoming interrupt is disabled to give the QR time to process all the data.
Coupled with the use of tasks as described in the previous section, this allows for lock free writing to and reading from the buffer. As an example consider the *comm_process_message* task, which does message decoding, validation and updating of the system. This task only reads the Rx buffer head index (to see if there is data in the buffer) but only updates the Rx buffer tail. Conversely the Rx ISR only reads the tail and updates the head. So there cannot be conflicting updates to variables.
A similar system is used for the link from QR to PC. The task manager calls the function *comm_send_current_state*, which then collects data from the system, formats it and put the data in a buffer. To make sure the data will be send, a call to the ISR is made. The Tx ISR then sends bytes from this buffer until it is empty.
To measure performance of the system, a measurement bench has been created. This test bench is described in the CPU features paragraph. Results of the timing data can be found in Table 1.

3.1.3 Cpu Features

To increase safety, overflow and divide-by-zero detection is enabled. The QR policy in such cases is to panic, since we don't know what is going on. There was some thought about more advanced schemes that could ignore the occasional overflow and panic only when they became abundant. However detection of such situations is not trivial and we where forced to scrap the feature due to time constraints.

As it turned out our application had no divide-by-zero errors because of lack of divisions, but there were overflow exceptions. Via the x32 frame pointer it is possible to retrieve the address of the instruction that caused the exception, however relating this back to a source file is quite involved. The assembler can produce a symbol table for debugging. However this would lead to much error prone manual looking in the debug symbol table. Another way to tackle the problem could be to use the provided debugger. However the provided debugger seems either very complex, or unable to handle multiple source files, we could not get it to work usefully.

Our solution to these problems is to take a more dynamic approach and "register" each function at start up. The address-of operator (&) of a function is defined at the linking stage, and filled in in our program. As such it is possible at runtime to build a table of all functions in the program including names and sizes. This enables the overflow ISR to print a stack trace of the location where things went wrong. As it turned out quite a few of the communication routines used signed values and where flipping the most significant bit, causing overflow. Also the task component had some signed/unsigned errors.

In addition to the arithmetic errors, another requirement is to provide function timings to ensure proper operation. The table of function addresses and sizes coupled with the x32's trap bit on instructions makes it possible to easily and accurately profile the x32 program during normal operation with very minimal overhead. This way it does not require adding code to functions and is transparent to the rest of the code.
Each second a function is selected from the table using round robin. The function gets a trap bit on the first instruction and on the (last) return instruction. In the trap ISR we can then use the nanosecond clock to time the function. The measurement overhead is tested using an empty function. This mea-

surement is used to correct all the timings. The table with statistics is printed out at program exit and is corrected for this overhead. The statistics include min, max and average timing, number of invocations and code size for each function.
This profiling strategy did however reveal an issue with the x32. The x32 gets stuck if it encounters a trap instruction while executing at an execution level higher than the level of the trap ISR. The trapped instruction cannot execute until the trap bit is cleared, but the trap ISR that will clear the bit will also not execute since its execution level is to low, resulting in a stuck x32. One solution is to make the trap ISR the highest possible execution level, but the priority of the trap ISR was carefully selected to be below that of ISR's that cause panic, since we never want to trap in such situations. A few critical sections caused this issue to present itself in our program. The problem was resolved by explicitly restoring the execution level after a critical section instead of the function return doing so implicitly. This ensures the trap ISR can run on function exit.

3.2 Ground Station

The team decided the ground station should have a graphical user interface. To support multiple OS's, there was an obvious need for a portable GUI platform, preferably without the need to install a whole range of runtimes or libraries. The final choice was C#, because of prior GUI experience in the team and standard deployment of runtimes on most operating systems. It showed that the models underpinning unix and windows GUIs differ radically in the face of multi-threaded applications. The chosen solution sacrifices some elegance by running an update loop with a timed delay and manual os message dispatching.
The ground station centers around the QR-Model class, a pc side model of the QR. This model includes for example: values reported by the QR and set points sent to the QR. The main window simply serves as a visualization of this model and updates this visualization periodically. The main window also manages setting up the communication and handles the keyboard input.

Providing a cross-platform system for the joystick input was slightly more involved. When initialized, the joystick component will enumerate and create a joystick based on the type of operating system. In the case of a *UNIX* system, the joystick will be based on a file system stream that is decoded. On *Windows* the joystick is polled through the *AForge* library. For testing without a physical joystick an operating system independent On-Screen Joystick is included. Attention was given to the correct usage of signs (i.e.: when pitching down the joystick should set a negative value, not a positive) and managing the scale of the output. The scale was corrected such that each joystick implementation provides values in the range of -1.0 to +1.0. On the QR this scale is converted by an exponential lookup table, which creates sensitivity around the center, and power around the maximum joystick angles.

Similar to the On-Screen Joystick, it also proved useful to be able to test the communications to the x32, without an x32 actually present. To this end, the x32-SIM tool was used to simulate the x32 code, and redirecting its standard input and output to the communication component of the ground station.

Because work on the ground station took less time then expected, there are some additional features. One of these is the artificial horizon, which visualizes the current attitude as reported by the QR. Another is a graph presentation of the time since the last full message was decoded successfully. This visualizes the update speed of messages sent from the QR.

3.3 Filters

3.3.1 Logging

In order to design the filters, it was necessary to find the signal properties of the sensor output values for different QR flight scenarios. The application is altered to enable recording of as much sensor output data as possible on the QR. The log files ranged in length from 12000 samples to 30000 samples, which corresponds to 8.4 sec and 21sec. After designing the filters it was possible to run the filters on the board and collect a log file which included outputs of filters implemented on the board.
Proper selection of the flight scenarios is very important for the correct design of the filters. Initially the QR has to be horizontal for a significant number of samples in order to compute the mean values of the signals. After that, the QR was brought to lift-of throttle and subjected to strong accelerations. That was done to ensure that the filters do not overflow while in operation. If the filters are dimensioned for accelerations that are too small, then the filters might either not respond fast enough or overflow. Another set of log files where recorded while subjecting the QR to accelerations that simulate, to the authors feeling, normal flight conditions.

3.3.2 C Simulator

The log files that were recorded are read by a C simulator. The C simulator reads the sensor data and runs the 4 filters for this input data. Then, it records the output of the filters in another file, which is imported to *Matlab*. Then the response of the filters which are implemented in *Matlab* code is compared to the response of the filters simulated in C. That was to make sure that the responses were the same.

3.3.3 Filter Implementation

The following filters were implemented :

- Kalman filter x, which receives as inputs the gyro y and accelerometer x and computes theta.
- Kalman filter y, which receives as inputs the gyro x and accelerometer y and computes phi.
- Butterworth filter gyro z : second order Butterworth filter, cut-off frequency 10Hz, filter gyro z
- Butterworth filter acc. z : second order Butterworth filter, cut-off frequency 10Hz, filters acc. z. [1]

Figure 2 shows the input signals to the filters, namely the signals from the three accelerometers and three gyros for our current log file. The next figure shows the integrated angle using the signal from the gyro compared to the relevant accelerometer. This figure was used to determine *p2phi*, which is equal to 0.0032. Figure 3 shows the output of the Kalman x filter, which computes theta. This figure depicts the theoretical output of the filter computed by Matlab, the output of the filter implemented in C, and output by the simulator. Furthermore, the picture presents in red and blue color the errors that come from division by the factors C1 and C2 due to limited number of fractional bits. It is obvious that division by C1 causes no loss of accuracy, while division by C2 causes a very small loss of accuracy. Figure 4 show the unfiltered gyro z signal and the filtered gyro z signal by Matlab (left figure) and the comparison of the matlab filter and the c implementation (right figure). It is obvious that the two filters produce exactly the same values.
Figure 5, shows some dangerous situations in filter design for critical applications. The graph on the left side illustrates a wrong gain value for the Butterworth filter. The graph on the right side demonstrates a scenario where the filter overflows only for big values measured by the z-axis gyro. As a result, if the

[1] This filter was not used in final implementation, because the accelerometer signal was finally not needed in the control equations.

log files that are used do not correspond to extreme real life flight scenarios, then the filters that the engineers design might cause safety critical problems.

3.4 Control functions

Depending on the chosen control mode, the control function uses as input either the input of the user at the ground station (in manual mode) or the difference between the input of the user and the corresponding output of the filters (in *yaw control* and *full control* mode) this is referred to as the error. To obtain a valid value for the error the inputs must be scaled to the same range. The appropriate ranges were found from analyzing the filter functions.

In *manual mode* the input is translated to individual motor settings (function output) using Equation 1. Where T, ϕ, θ and R correspond to input throttle, roll angle, pitch angle and yaw rate respectively. The coefficients A, B and C are tuning variables for determining the effect of change in input variables on the quad rotor. The tuning variables are set by trail an error until a good compromise is found between responsiveness and accuracy. The exact value for such a compromise depends on the users preference and the type of flying maneuvers to be performed.

$$\begin{bmatrix} U_1 \\ U_2 \\ U_3 \\ U_4 \end{bmatrix} = \begin{bmatrix} T & \theta & -r \\ T & -\phi & r \\ T & -\theta & -r \\ T & \phi & r \end{bmatrix} \begin{bmatrix} a \\ b \\ c \end{bmatrix} \quad (1)$$

Equation 2 shows how the errors are translated to individual motor settings in full control mode. Where, $K_{P_{\theta,\phi}}$ is the proportional controller gain for pitch and roll and K_{P_r} is the proportional controller gain for yaw rate. The gain values are determined using the *Ziegler-Nichols* method which means increasing them individually until the controller becomes unstable (ultimate gain). Instability becomes apparent when the quad rotor starts to oscillate increasingly. The gain value is then set to half the value of ultimate gain.

$$\begin{bmatrix} U_1 \\ U_2 \\ U_3 \\ U_4 \end{bmatrix} = \begin{bmatrix} T & \theta_{err.} & -r_{err.} \\ T & -\phi_{err.} & r_{err.} \\ T & -\theta_{err.} & -r_{err.} \\ T & \phi_{err.} & r_{err.} \end{bmatrix} \begin{bmatrix} a \\ K_{P_{\theta,\phi}} \\ K_{P_r} \end{bmatrix} \quad (2)$$

In *yaw control* mode, the throttle, pitch and roll are controlled using the corresponding parts of Equation 1 and using the user input. While the yaw rate is controlled using the yaw rate part of Equation 2 and using the yaw rate error as input.

In the aerospace field the common practice is to determine the proportional controller gains and the tuning variables mathematically using the physical specifications of the platform and the equations of motion linearized around the operational point. This is done to prevent costly accidents during initial testing. But such approach was not possible due to lack of time and the unavailability of the quad rotors physical specifications such as mass and moments of inertia.

Another point where this implementation differs from common practice is the use of a pure proportional controller instead of a *Proportional-Integral-Derivative* controller. Although the proportional term contributes to the bulk of the change in a PID controllers output, it is insufficient by itself because it is not sensitive/responsive enough to small errors and it does not compensate for steady state errors. This was explained to the lab tutors in the beginning of the course and the implementation of a *proportional-derivative* controller for roll and pitch was suggested by the group. But the tutors maintained that the proportional term is sufficient for the purpose of this lab. Unfortunately, it became apparent in the last lab session that what the tutors meant was to use two proportional controller loops one for angle and one for angular rate. The result of that misinterpretation of tutors advice is that the current implementation of the controller is far from perfect. Al-

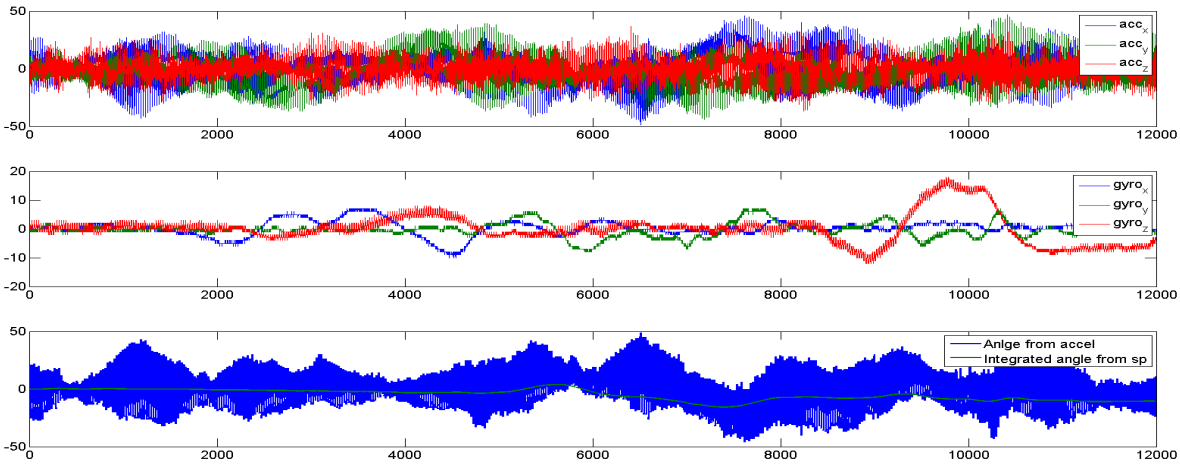

Figure 2: Input signals (raw data) from the sensors to the filters.

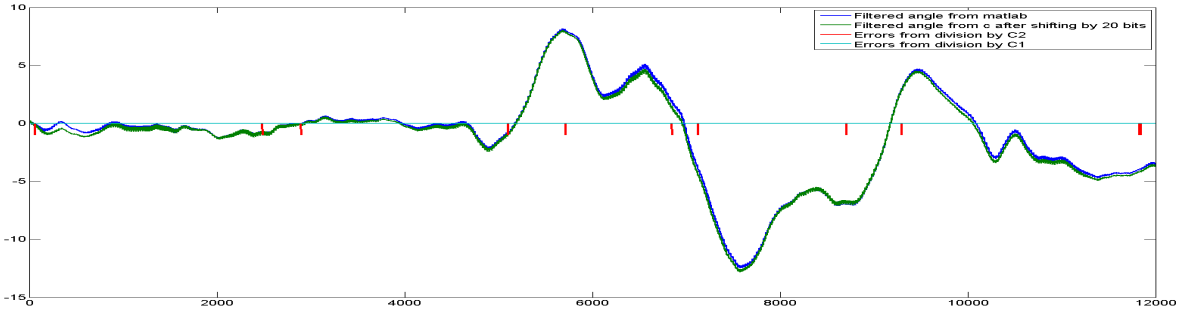

Figure 3: Filter output of the Kalman x (theta) filter.

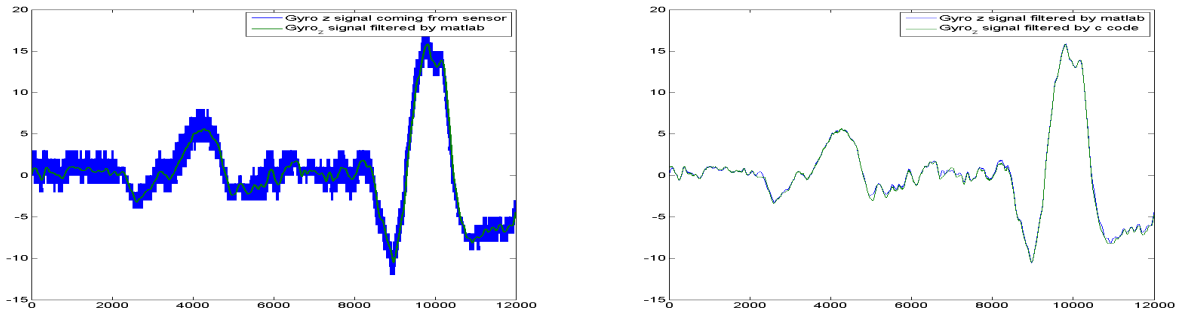

Figure 4: Unfiltered and filtered (Butterworth) gyro z data.

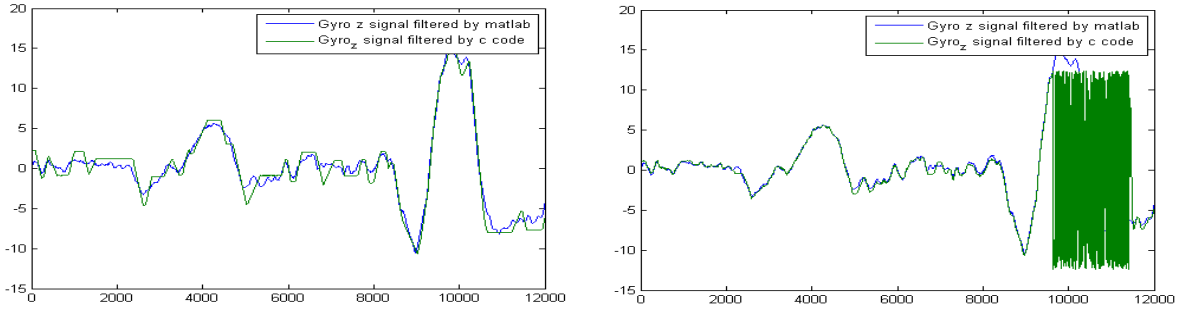

Figure 5: Results of wrong gain value and clipping due to overflow.

though the full control mode provides a large improvement to controllability over the manual mode it is "far from easy" to control.

4 Results

Table 1 shows timing performance of the system and its functions. The quad rotor responds fast to joystick inputs, wire and wireless. The yaw controller damps yaw movement. In *full control* mode the system oscillates and the yaw controller becomes unstable. Pitch and roll control do not provide strong correction.

name	max	avg	size
cpu_empty_function	335	1	8
comm_send_current_state	433	430	362
comm_write_char	69	68	162
isr_primary_tx	35	22	206
comm_process_message	343	8	3272
isr_timer1	334	331	570
controllers_update_controllers	277	235	1740
controllers_update_setpoints	14	13	190
isr_qr_sensors	583	554	490
but_gyro_z	54	53	376
but_acc_z	53	51	362
kalman_filter_x	66	65	456
kalman_filter_y	66	65	456
sensors_callibration_step	148	91	1192
motor_clip	36	28	360
motor_set_throttle	174	161	310

Table 1: Timing overview of the main functions.

5 Conclusions

5.1 Design Evaluation

By using an interrupt-driven system for the time-critical part and a basic task manager the system is a good and stable platform with fast timing results and flexibility for extension. The communication system is basic and limited due to the fixed format of the message, but supports all the data necessary for the current application. The application of Kalman filters give stability to raw sensor data with noise, although difficult to fine tune. The quad rotor does correct for error in pitch, roll and yaw movement, but is not stable enough to allow free flying on the full control loops. This is because the lack of a rate control loop which runs over the angle control loop. The yaw control works good but would work better if it was possible to apply learned designing techniques for control loop design. Most of these techniques are not used because of unknown and missing data/units.

5.2 Team results

As a team we worked together, combining knowledge to overcome problems during the project. Because of the advanced framework using a task manager, bugs sometimes occurred during lab sessions. These bugs delayed the progress during sparse time, and so delayed overall progress. To change this, the team decided to add more time to extensively test the system before lab sessions. By working long days and evenings we managed to get back on schedule, resulting in a finished design at the last lab session including wireless comm's.

5.3 Learning experience

As a team we combined knowledge of each of the individual members as good as possible. The mix between *Computer Engineering*, *Embedded Systems* and *Aerospace Engineering* was very educating because of different viewpoints to approaches, and the combination of sciences. We are sure this project increased interest in the fields *Embedded Systems* and *Aerospace*, a strong point for the university. Because of the application in practice of theory which is obtained during classes, this project is close to real world engineering. For most of us this project is the first application of education in reality.

6 Appendices

- Figure 6: Interface diagram (public/private methods and variables).

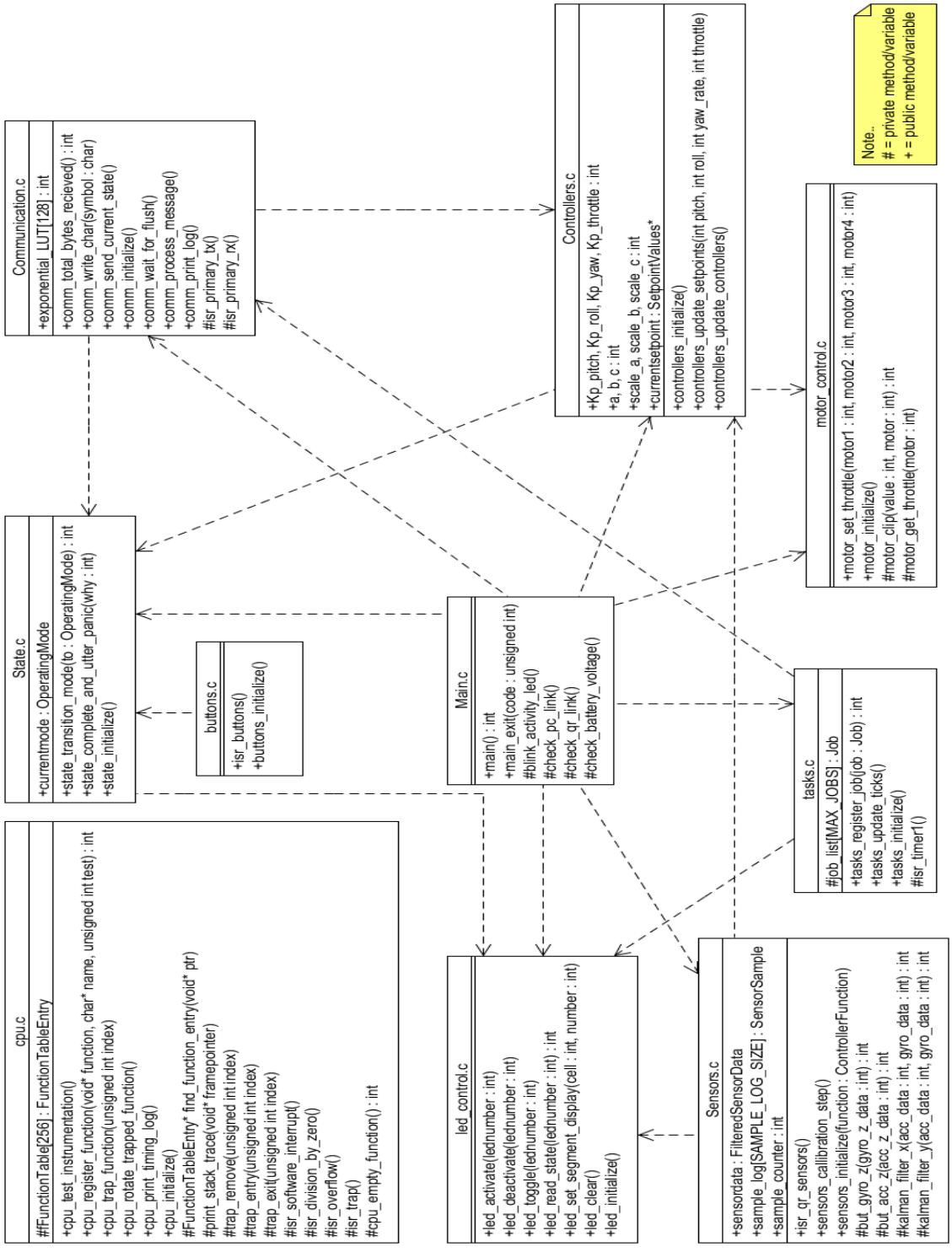

Figure 6: Overview of the different components with their respective interfaces.