CS4140 - Embedded Systems Laboratory

Tim van der Horst
4437837

Vibhav Inna Kedege
4998596

Hao Liu
4807456

Tanmay Manjunath
5082501

ABSTRACT

In this report, we present the system architecture and approach used to create a quadcopter flight control system for the quadrupel system. By using a Round-Robin with interrupts architecture combined with a simple controller and reliable communication protocol, we created a system which runs at 100Hz in DMP mode (with filtered sensor signals) and 400Hz in RAW mode (with unfiltered sensor signals). The quadcopter was able to achieve stable flight and could be controlled via a wired or wireless medium.

1 INTRODUCTION

The objective of the project is to design and implement software for the Quadrupel Flight Control Board **(FCB)** in order to create a dependable, robust and most importantly safe system which is able to achieve stable control of a quadcopter.

The hardware has memory and processing power constraints but should still be able to interface with joystick and keyboard peripherals in real-time.

In this report we will explain the design choices we made, how we implemented our design and experimental results. We will also evaluate our design and give areas of improvement.

2 ARCHITECTURE

2.1 Overview

The system architecture can be divided into two parts: hardware architecture and software architecture. The diagram in Figure 1 shows the various hardware components and how they interface with each other. In the next section, we will explain our design choices for the software architecture.

2.2 Software Architecture

The software architecture is shown in the Figure 2.

The architecture on the FCB side uses a Round-Robin with interrupts architecture. We investigated a number of architectures such as a Real-Time OS, non-preemptive priority based scheduling and round-robin with or without interrupts.

The benefits of using RTOS are easy creation of tasks and access to a preemptive priority-based scheduler which could lead to a lower latency if tasks are designed properly. In addition, FreeRTOS has already been ported to the ARM Cortex-M architecture [6]. Despite these advantages we decided that we would have to invest too much time in porting drivers and other existing code to this architecture. The code footprint is also quite large.

The simplest architecture we evaluated was the round-robin without interrupts architecture. The advantages are that it is very easy to implement and analyze. However, all tasks have the same priority which may become problematic if some tasks take more time than expected. The response time can be improved by using interrupts. Immediate needs are serviced in the interrupt handler and a flag is set which indicates that the rest of the task needs to be completed as soon as possible. We decided to implement this architecture as it is simple, easy to analyze and not very error-prone. It is also easy to extend with other features such as a priority queue used to service tasks based on priority if that is found to be necessary.

In addition, we can make use of the NRF-SDKs built in timer interrupts to create periodic tasks which can run at different frequencies. The timer interrupts simply set a flag which indicates that the task is ready to be serviced, which happens in the main loop which runs at the processor's clock speed.

Figure 2 shows how we implemented the round robin architecture. Based on the requirements, we created a number of tasks which needed to run at different frequencies. Some tasks are more important and need to run at higher frequencies (control algorithms and IMU reading) than others.

The PC side also uses a round robin architecture with interrupts (but without timer interrupts). When a keyboard button is pressed, an interrupt is raised and immediately forwarded to the FCB to achieve the lowest possible latency.

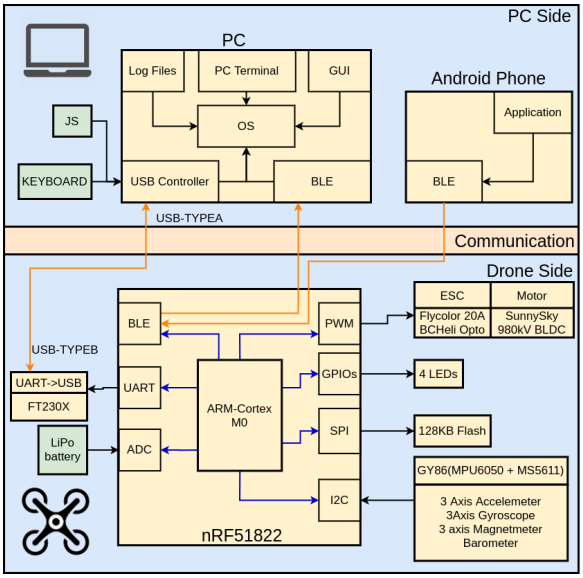

Figure 1: The architecture of Hardware

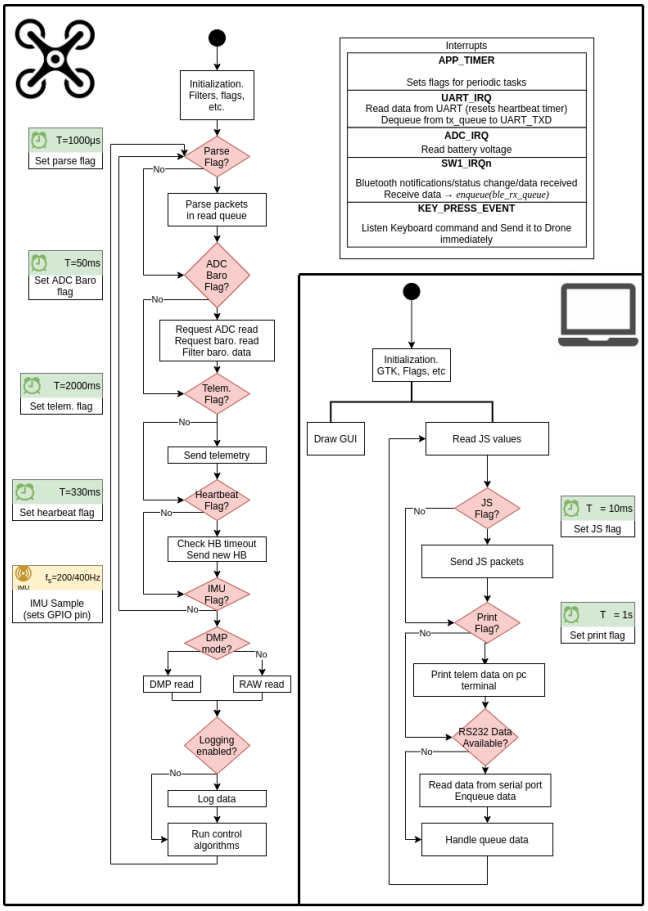

Figure 2: The software architecture

3 IMPLEMENTATION

3.1 Contributions

3.1.1 Tim van der Horst. I created the complementary filter code for RAW mode, calibration code, Android application, wireless control, heartbeats, round-robin with interrupt architecture, parsing packets from the input queue on both PC and FCB side. I created the control code for safe mode, panic mode, manual, yaw control, full control and height control. I created the state machine used to keep track of the current mode. I wrote the initial version of the GUI, which was later completed by someone else. I also designed the compression scheme for logging although this was ported to C by someone else. I also wrote the angle calculation code for RAW mode (both fixed and floating point versions). I also did all profiling except for the queuing delay profiling.

3.1.2 Hao Liu. Firstly, I implemented the joystick data reading and made sure that the joystick data was accessible both on the pc and drone side. Secondly, I implemented the logging for writing data into the flash and the functionality to read data from flash into a file via RS232. Thirdly, I worked on improving the GUI which included using the cario library to draw gauges, using the pango library to add numbers on the gauge and added error textviews, pressure indicators etc. At last, I helped in turning the parameters for yaw control, full control, height control and raw mode.

3.1.3 Vibhav Inna Kedege. I predominantly worked on the the control of the drone and signal filtering. In terms of control, I wrote the initial code for YAW mode and FULL mode which was later on modified by someone else. Once the controllers were implemented, I worked on tuning the P controllers for the above 2 modes. In terms of the filters, I wrote the code for the kalman filter according to the implementation given in [13]. As we chose 3 possible filters to implement in RAW mode, I worked on tuning the coefficients for each filter methodically in order to find the filter that gave the best result for rolling and pitching.

3.1.4 Tanmay Manjunath. I worked on making the communication protocol more reliable. This included cyclic redundancy code for error checking of packets and escape packets to escape data packets with same value as one of the header packets. In addition, I created a separate queue for joystick packets with last-in read to reduce the communication time of joystick packets. I created a state machine to parse the two queues (Joystick queue and Keyboard+Heartbeat queue) and to keep track of escape packets. However, as the packet corruption rate was low, we later decided to stick to a simple communication protocol to keep the parsing delay to a minimum. Also, I implemented the run-length encoding scheme for logging in C and aided in debugging the raw mode.

3.2 Mode State Machine

An important aspect of the system is the ability to change mode quickly, reliably and safely. The various modes along with the transition action has been shown in figure 3. There are various safety requirements related to the working of the quad rotor that have been captured - such as returning to safe mode between mode switches, panic mode always being reachable, etc.

We decided to implement the modes using a state machine which uses two arrays of function pointers which are called when leaving a mode and entering a new mode. The first function call checks if the transition to a new mode is valid.

Whether a transition is valid depends on the requirements, the current mode and the new mode. Leaving safe is only valid if the joystick is in neutral position for example. Actions which need to be performed (such as setting the keyboard lift trim to 0) can be performed in this transition.

The next function pointer which is called is the `enter_mode_action`. For example, when calibration mode is entered the current variable sums need to be reset to 0. Entering panic mode will start the panic mode countdown timer.

This design makes it easy to extend mode transitions with more actions and makes it very simple to add additional modes.

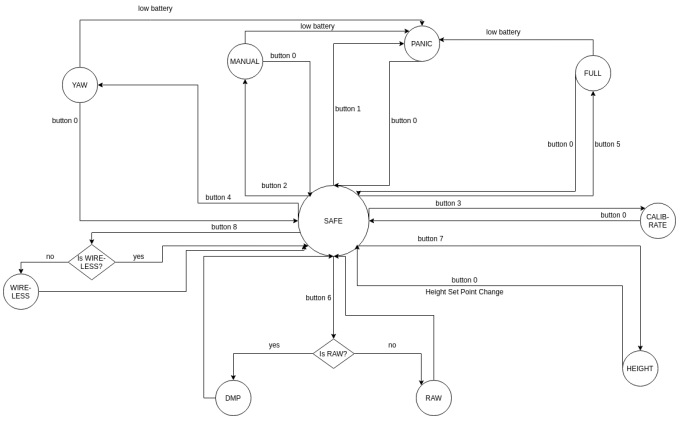

Figure 3: Mode State Diagram

3.3 Communication

The communication protocol was designed to be dependable and simple to implement. The packets which are sent over the PC link have a fixed width of 8 bits. We decided to keep all of our packets byte aligned to reduce the complexity of the parsing code and reduce misalignment errors.

Our packet structure has the following simple format:

| Header | Data | Data | ... |

Each packet starts with a header which indicates the type of packet. The header is followed by 1 or more bytes of data. Packets have a fixed length and each packet type may have a different length. Packets are typically very short, between 2-4 bytes including the header.

We chose not to encode the packet length because this information is already implied by the packet header.

We evaluated whether a checksum such as CRC is necessary to check if packets have been corrupted. The packet corruption rate was extremely low, so this was not deemed to be necessary. An initial version of the design also included a packet counter and an ACK for every packet. However, the packet loss rate was also extremely low so this was also removed to reduce the parsing complexity.

An important aspect of the drone is dependable communication when it is being controlled using the joystick. The position of the joystick is encoded using a 16 bit integer for each of the axes and a 1 bit value for each button. The axis values can be communicated to the FCB using absolute values or delta values (difference between previous and current position).

The advantage of delta packets is that they can be smaller as the difference between two joystick readings is quite small. However, if delta packets are dropped unexpectedly or corrupted, the values stored on the FCB will be different from the actual position of the joystick. This can be corrected by occasionally sending absolute values in combination with delta values. For safety reasons we chose to stick with absolute values despite the larger packet size.

An important safety feature of the drone is entering panic mode if the PC link becomes erratic or packets are dropped. We chose to implement this using heartbeat packets which are sent periodically by the FCB.

When the PC receives a heartbeat packet, it responds with an acknowledgement. Whenever the FCB receives data via UART the heartbeat timer is reset. If too much time has elapsed since new data has arrived, the FCB will enter panic mode.

Although the heartbeat packets are not really required when controlling the drone via cable (as joystick packets arrive at a high frequency, which also resets the timeout) these packets are required in wireless mode to ensure a minimum packet arrival period due to the reduced amount of communication (see section 3.9 for more details).

3.4 Parser

A parser was designed on the FCB side to decode the incoming packets. The implementation of the same is shown in Figure 4.

The parser code first checks for the length of the queue to determine if there are sufficient data bytes to continue with the parsing. Then, the top of the queue is checked for a valid header. Since the length of the packet and the type of data is implied by the header, the following data bytes in the queue are processed depending on the header value.

An important aspect that we thought about while designing the parser was to avoid the queue length from becoming too long as this would lead to larger queuing delays if a packet is corrupted. To solve this a threshold is set on the length on the queue. If the queue length were to reach this threshold, all the packets in the queue are discarded. The disadvantage of this is that some important packets may be lost. However, we have found that at a threshold of 20 packets are rarely lost.

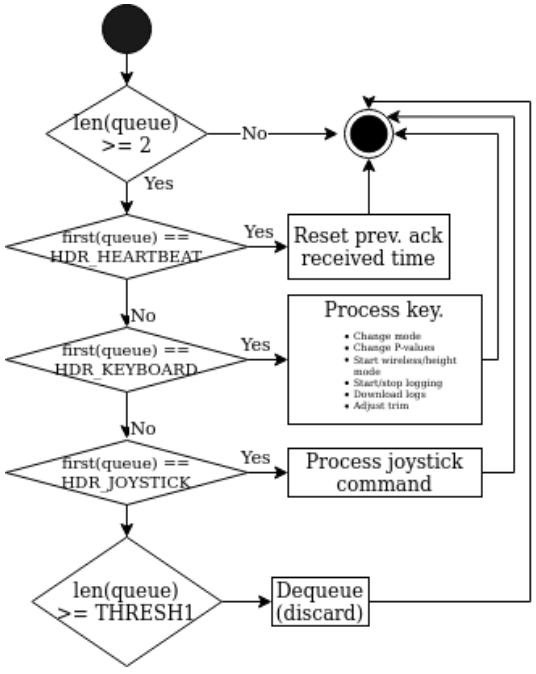

Figure 4: Parser Implementation

3.5 Logging

Data logging is designed for analyzing the system performance and verifying the filter design. Logging is not enabled all the time - only when necessary. It runs at the same frequency as the control loop. Logging can be started via a keyboard command and stopped at any time. Logging also stops when the flash is full.

Resetting the logs and downloading logs can only be done in SAFE mode as this interrupts the control loop for quite a long time.

Log packets have a fixed length and format. Variables which need to be logged are encoded and written to a buffer and then the log packet is written to flash once all variables have been added to flash. The address where the packet will start is incremented by the packet size every time a log packet is written to flash.

Each byte takes 15μs to write to flash, which meant that we needed to compress the data as much as possible to reduce the time spent logging. This was achieved by logging delta packets (changes in variable) and scaling this delta packet so it fits in one byte. The first packet that is recorded are absolute values of each of the variables, all subsequent packets are deltas. An error term is maintained for each variable so that errors in deltas which occur due to integer scaling do not accumulate but always stay close to 0.

3.6 Calibration

Calibration was implemented as a separate mode, similar to safe, yaw, full control, etc. Motor RPM is always set to 0 during calibration. For each of the variables that needed to be calibrated, a 64 bit sum variable was initialized to 0. When calibration mode is enabled, the current sensor output is added to the sum and a counter is incremented. When calibration mode is left, the mean value of the variable during calibration is calculated by dividing the sum by the counter. This is subsequently subtracted from the sensor value to obtain a zeroed output.

Although 64 bit arithmetic is slower on a 32 bit architecture, we decided to use this to prevent overflows from occurring as overflowing a 64 bit value takes at least 2^{31} additions of a 32 bit value, which will take about 62 days at a loop frequency of 400Hz.

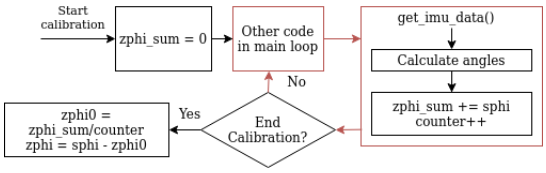

Figure 5: Calibration mode diagram. Main control loop code is highlighted in red

3.7 Controllers

In order to implement the controller, the same control loop was used in both DMP and RAW modes. Figure 6 shows how the control loop was implemented.

The difference between the RAW and DMP mode was the source of the sensor signals. In the case of DMP, the sensor values were already filtered and thus could directly be used in the control loop. However, In the case of RAW mode, due to the large amount of noise captured by bare gyroscope and accelerometer reading, an additional filter unit was added following which filtered values were sent to the controller.

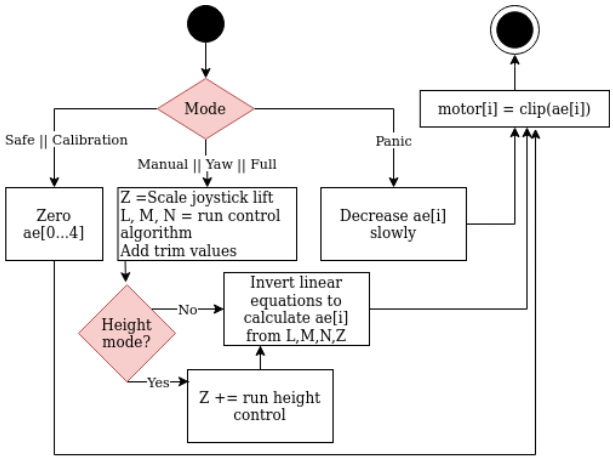

Figure 6: Control loop diagram

For the controller, we consider the control loop diagrams that have been mentioned in [13] where a single a single P controller is used for yaw control while 2 controllers, one to correct rate and the other to correct angle are used for pitch and roll control. The P values that gave the most stable behaviour can be seen in table 1. We are not sure why the P2 values have opposite signs, but negating any of the values makes the system unstable.

Controller	Parameter	Value
Yaw	P	-800
Pitch	P1	0.4
	P2	111
Roll	P1	0.4
	P2	-111

Table 1: Controller Parameter-Values

3.7.1 DMP Mode. On activating the DMP mode, the InvenSense sensor gave out filtered accelerometer, gyroscope and angle values. Before beginning the control, the sensor values were first calibrated using the method described in section 3.6. Following this zeroed values of sensor values were used in the YAW and FULL control modes.

3.7.2 Raw Mode. In order to enable RAW mode in which bare accelerometer and gyroscope values are used, the sensor values had to first be filtered to give a less noisy signal with lower variance and also fused in order to obtain the angle values. A 2nd order butterworth filter was initially used for the gyroscope values for yaw control in order to give a smoother sr value. However it was later found that the gyroscope values proved to be stable enough after calibration and therefore this filter was removed and the calibrated sr signal (zr) was used. In order to obtain stable angle values, sensor fusion techniques were used to combine the estimates of the angles

produced from the accelerometer and gyroscope readings. To do this, 3 sensor fusion techniques were tested:-

(1) Kalman Filter
(2) Kalman Filter with noise co-variance
(3) Complementary Filter

The first filter that was used was the kalman filter implementation which is as given in the [13]. On exploring on the fundamentals that govern this sensor behaviour, it was found that an extra noise co-variance term was found in a few implementations [12] [4]. Thus, this second filter was also incorporated into this work and analysed.

Apart from the kalman filter variants, a complementary filter [1] was also a filter that was considered as a possible choice of filter due to its simplicity of tuning and implementation. The following equation governs the behaviour of the complementary filter [2]:-

$$\phi = K_{gyro}(\phi + z_p \cdot dt) + (1 - K_{gyro})z_{phi} \quad (1)$$

The roll filter has the same equation, with z_{phi} replaced by z_{theta} and z_p replaced by z_q.

A complementary filer is equivalent to combining low pass filtered accelerometer data with high pass filtered and then integrated gyrometer readings. Complementary filters eliminate drift which occurs when integrating the gyrometer readings by combining these readings with accelerometer readings - which have high noise due to motor vibrations but give accurate values in static conditions [13] [2]. The filter is controlled by the K_{gyro} and $K_{acce} = 1 - K_{gyro}$ parameter. In our experiments it was found that a stable output angle was obtained when $K_{gyro} \geq 0.9$.

To compare the filter output, data from the sensors were recorded and plotted along with the output from each sensor. The frequency at which the control loop was running was 400Hz and the FCB was tilted about all the 3 axes. The values of the constants that worked the best have been shown in table 2 and the corresponding graph has been shown in figure 7 for phi and 8 for theta.

Filter type	Constant	Value
Kalman Filter (1)	C1	10
	C2	100
	P2PHI	0.0025
Kalman Filter (2)	C1	0.01
	C2	0.001
	P2PHI	0.0025
	R	1
Compl Filter (3)	K_gyro	0.9
	K_acce	0.1
	dt	0.0025

Table 2: Filter Parameter-Values

In table 3, the delay and cut off for each type of filter has been measured. The delay in each case was measured by subtracting the time of the peaks of the input data and the filtered output data, and multiplying this value with the time between subsequent samples ($1/raw\_frequency$). For the cut off frequency, a transfer function between the input angle and output computed angle was computed for each filter using the System Identification Toolbox in MATLAB [3]. For example for the kalman roll filter it was computed between sphi and phi (kalman output). Subsequently, the bode plot was drawn for each transfer function and the 3dB frequency point was recorded.

Filter variant	filter type	delay (ms)	cut off (Hz)
Kalman Filter (1)	roll	12.5	7.1
	pitch	5	7.7
Kalman Filter (2)	roll	15	7.3
	pitch	7.5	7.3
Compl Filter (3)	roll	12.5	7.2
	pitch	7.5	6.8

Table 3: Filter delay time and cut off frequencies

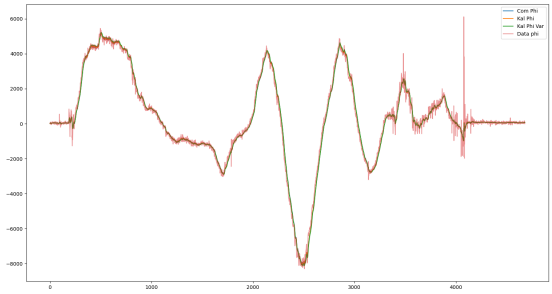

Figure 7: Filter output - phi

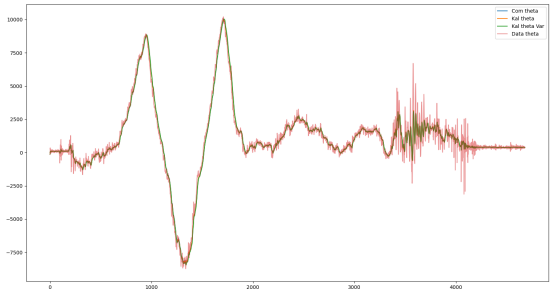

Figure 8: Filter output - theta

From table 3 and the graphs, it can be seen from the graphs that these parameter values are able to track the data signal in a similar manner with almost equal delays for the estimation of ϕ and θ. Therefore to choose the right filter and possibly tune the coefficients for the real world scenario, the filters were made to run on the FCB. On testing this, it was found that with both the kalman filters (with or without co-variance) the drone was still showing unstable "twitching" behaviour. This was even true with the values

given in table 2. In the case of the complementary filter, for the values given in table 2 there was a unstable behaviour seen on the drone. The parameters of the complementary filter were tuned further and it was found that on keeping a value of $K\_gyro = 0.999$ and $K\_acce = 0.001$ the drone became stable. Due to this, the complementary filter with these parameters were selected.

We think this unstable behavior may be related to the inverted sign in the cascaded P controller. Unfortunately, we only realised this during a code review before the final demonstration.

3.8 Height control

Height control was achieved by filtering and combining barometer readings and accelerometer readings. Pressure readings can be converted to change in altitude using the following relation:

$$P_1 = P_2 e^{\frac{-\Delta H}{cT}} \tag{2}$$

This equation was linearized for standard room temperature and altitudes near sea level:

$$\Delta H = \frac{sP_1, sP_2}{-0.12} \tag{3}$$

This linearization gives an error of approximately 0.01% for altitude changes of 4m near sea level. This error is much lower than the accuracy ($\pm 1.5mbar (\approx \pm 12.5m)$) and resolution (0.012mbar (10cm)) of the sensor.

Unfortunately, this low accuracy means that the barometer alone cannot be used to maintain a stable altitude. Pressure readings were found to vary a lot indoors depending on the air flow over the sensor, position in the room and even outdoor weather effects. The following figure 9 illustrates the sudden spikes in readings. The actual altitude varied between 0-1m of the starting value.

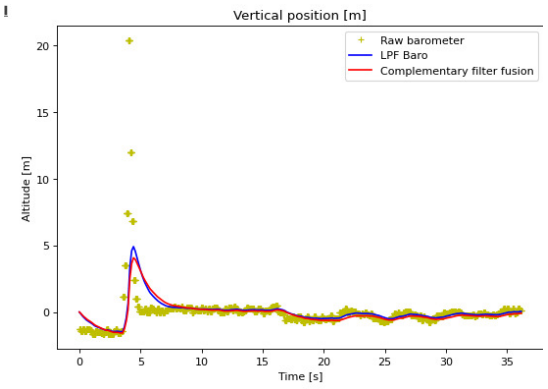

Figure 9: Spike in barometer pressure readings

To compensate the inaccurate barometer values we decided to opt for sensor fusion with the accelerometer readings, using a similar method as Klas Löfstedt (2018) [9].

The IMU measurements are performed in the body frame of reference and were first rotated to the inertial frame. In MPU mode, this was done using the measured quaternion readings as follows

$$sa_{z,I} = sa_{z,B} - (a^2 - b^2 - c^2 + d^2) \cdot S \tag{4}$$

In this equation, $sa_{z,I}$ is the measured z-component of the acceleration in the inertial frame. $sa_{z,B}$ is the raw accelerometer reading (body frame). a, b, c, d are the quaternion readings and S is the sensitivity. After subtracting the gravity component we are left with the linear acceleration of the quadcopter in the inertial frame, which we can integrate twice to estimate the vertical position.

This position can be combined with the barometer readings using a Kalman filter or a complementary filter. After implementing both, we chose to use a complementary filter due to the decreased complexity. The block diagram can be found below

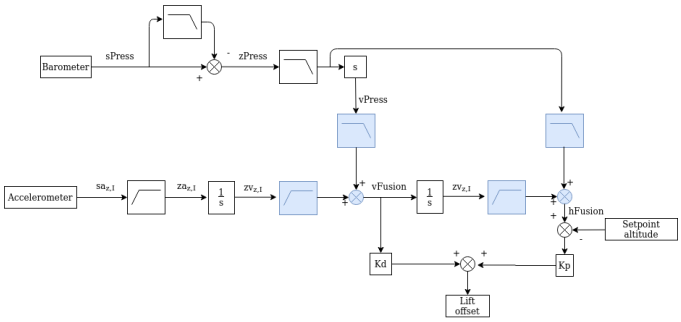

Figure 10: Height control block diagram

The raw barometer and accelerometer readings are first high pass filtered with a very low cut-off frequency (about 0.1Hz). Calibration on a non-horizontal surface will cause the accelerometer readings to be non-zero on average, which will have a big impact on the position after integrating twice. The high pass filter allows for constant calibration of the accelerometer readings. The same applies to the barometer readings - during testing we noticed that there can be considerable drift in pressure readings at constant altitude. We think this is due to changes in temperature, supply voltage or weather. The high pass filter removes some of this drift.

We then integrate the accelerometer readings to get the current velocity in the inertial Z axis, and differentiate the low pass filtered barometer readings to get the current change in altitude. This data is combined in the first complementary filter (center of the diagram, marked in blue). This gives a reasonably accurate velocity which uses accelerometer readings to quickly respond to changes while using barometer readings to remove any drift. This velocity data is then integrated again and combined with the absolute barometer reading to obtain the current position. The velocity and altitude are fed into a PD controller which are then used to control the height.

The complementary filter equation has two tunable coefficients: $\alpha_{velocity}, \alpha_{position}$. The two filters have the same equation as in Equation 1, except with different sensor inputs and coefficients. The higher the alpha values, the more the integrated accelerometer readings are "trusted".

Unfortunately there is a considerable delay between the movement of the FCB and changes in pressure readings (approx 1s) and an additional delay due to the low pass filtering of the pressure readings (another 0.5s). The integrated accelerometer readings are also less reliable at lower velocities. This delay is clearly visible in figure 11 at around t=163. The complementary filter output (green line) starts to increase almost a second before the raw pressure

readings start to increase. By combining the two sensor inputs we have therefore managed to decrease the response time compared to using only raw pressure readings.

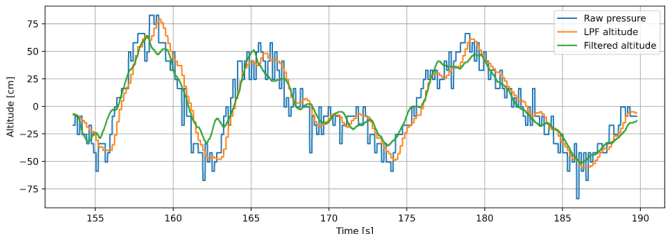

Figure 11: Height control with $\alpha_{velocity} = 0.995, \alpha_{position} = 0.98$

3.9 Wireless Control

The wireless control feature was completed by creating an application on Android which was able to communicate with the FCB. Android has better and easier to use BLE libraries than Linux, which is why we chose not to do wireless control via a laptop. We used the open source SimpleBLETerminal app [7] and ported the PC terminal code to Java to be able to communicate with the drone. The app interface can be found in Appendix A. It includes a joystick button, buttons to adjust trim, a slider for lift, buttons to switch mode, a panic button and telemetry data to monitor battery voltage.

On the FCB wireless mode was implemented using two separate states variables: the state of the connection is tracked using the connection handle pointer `m_conn_handle` and the state of wireless control is tracked using a global variable called `current_wireless_state`. The connection handle was already included in the BLE driver. The diagram in Figure 12 shows the process of wireless control.

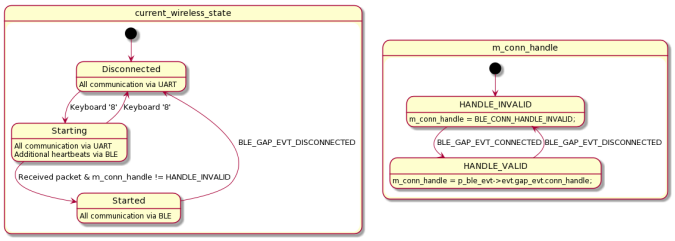

Figure 12: Wireless control state diagram

The FCB begins in `WIRELESS_DISCONNECTED` state. A user can connect their phone at any point to the FCB via BLE. This will cause the `m_conn_handle` to become valid. A user can then press the keyboard command to switch into wireless mode. The FCB will enter `WIRELESS_STARTING` mode and will start to send additional heartbeat packets via Bluetooth. In the meantime, communication will continue as usual on the UART channel. On the phone, an incoming heartbeat packet will trigger a response which is sent to the FCB. When data is received, this will cause `current_wireless_state` to enter `WIRELESS_STARTED` mode. This causes the parser to read from the `ble_rx_queue` instead of the UART queue. It also causes data to be written to `ble_tx_queue`.

Bluetooth packets are sent via BLE notifications and have a maximum size of 20B. Heartbeat packets are sent immediately to prevent heartbeat timeouts, while other packets are queued until there are at least 20 bytes in the `ble_tx_queue` to reduce overhead due to calls to `ble_send`. Since there is limited space on a phone display, we also chose to reduce the amount of telemetry sent via Bluetooth. Communication was further reduced by only sending joystick packets when the position of the joystick (or lift) changes.

For safety, the FCB will enter wireless control mode if and only if two conditions have been met: data has been received via the Bluetooth connection (which implies that there is a valid connection handle) and the keyboard command has been pressed in safe mode to switch into wireless mode.

If at any point during Wireless mode connection is lost, the FCB will revert back to the initial `WIRELESS_DISCONNECTED` state and the connection procedure has to be repeated. Communication via UART will resume if the cable is still connected.

3.10 Fixed point calculations

In RAW mode, all filtering was done using fixed point calculations. We decided to use a library, libfixmath [5] to perform calculations as it has support for trigonometric functions and is only a few KB. Libfixmath uses Q16.16 format (16 integer bits and 16 fractional bits) which means that the resolution is 2^{-16} and the values range from $[-2^{15}, 2^{15} - 2^{-16}]$. This gives ample resolution and range to perform all calculations.

The only problems occur when calculating angles in fixed point from accelerometer readings when the angles are small (stable hovering conditions). The lower resolution (compared to floating point) causes errors up to 10% when calculating the arctangent of angles using libfixmath. These errors were compensated by logging accelerometer data while moving the drone around and calculating fixed and floating point angles. A piecewise linear function was fit on the error, which reduced the error to 2% which was deemed acceptable. The additional execution time was small compared to the arctangent (3-5% of the execution time).

4 EXPERIMENTAL RESULTS

4.1 Code size

The size of the compiled code was 52160B of text, 244B of initialized data and 3232B of uninitialized data.

4.2 Code block latencies

The code was profiled on the FCB using simulated joystick input. If the mode was set to SAFE, all joystick values were zeroed. In any other mode, random integer values between [INT16_MIN, INT16_MAX] were sent to the FCB for each of the joystick axes.

The FCB was profiled in SAFE mode and FULL mode in both DMP and RAW mode. In addition, the latency of floating point angle calculations were compared to fixed point calculations. The execution time and period between each function call was logged to flash and downloaded later for analysis. The results can be found in Figure 14.

The results show that the execution of the RAW mode angle calculations are 2x as fast as the DMP calculations. In addition, the fixed point and floating point calculations are almost exactly the same speed. We used qfplib to perform floating point calculations, which is a floating point library optimized especially for the ARM Cortex-M0 architecture. The fixed point library we are using, libfixmath, has not been optimized for this architecture which could explain the similar execution times (as we would expect fixed point arithmetic to be faster on an architecture without an FPU).

IN MPU mode, the control frequency was set to 200Hz. In RAW mode, this control frequency was increased to 400Hz. Note that this is higher than the value we said during the final examination. We thought it was twice as low due to a bug in the logging code which caused only half of the packets to be logged.

In RAW and DMP mode we need to be able to complete all tasks withing 2500us and 5000us respectively to prevent deadline overruns. Since we are using a round robin architecture, the worst case execution time will occur if all timer flags have been set and all tasks need to be run concurrently. This will lead to a 99th percentile WCET of 4093ms and a 99.9th percentile WCET of 4859ms. Both fall within the 5000us limit, meaning we can achieve a loop frequency of 200Hz.

In RAW mode, the sum of 99th percentile WCET = 2987us and 99.9th percentile WCET = 3738us. Unfortunately, this means that there can occasionally be deadline misses when running in RAW mode (2500us is the limit for 400Hz). We decided to investigate this further by counting the number of significant deadline misses in RAW mode (where the period between the start of the control block is at least $WCET_{99} = 2987us$). This occurred in approximately 2.1% of the cases, which is slightly higher than the expected 1%. A deadline miss of more than $WCET_{99.9} = 3738us$ occurred in 0.03% of the cases. Figure 13 below shows a summary of deadline misses. It is clear that there are far fewer deadline misses in DMP mode.

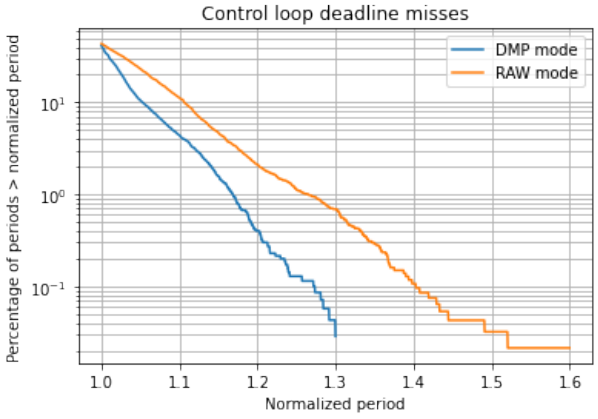

Figure 13: Deadline misses in MPU and RAW mode. Normalized period is defined by $\frac{T}{T_{setpoint}}$, where T is the time between two subsequent function calls

A deadline miss is not immediately a cause for concern, especially when it does not happen often. In addition, the mean period was still very close to the setpoint of 2500us which means that overall the FCB runs close to the setpoint control frequency. A deadline miss will cause the control loop to run slightly later than it is supposed to. It will cause the drone to rotate further towards the setpoint than expected. If multiple deadline misses occur after each other, the drone could overshoot the setpoint or become unstable. This was not the case during testing, which means that a frequency of 400Hz is probably fine - but we will not be able to go much higher than that.

A major source of delays comes from I^2C reads which are blocking. Reading quaternion sensor output from the IMU takes $950 - 1050\mu s$, in RAW mode this drops to $450 - 550\mu s$ and reading barometer data takes approximately $150\mu s$.

4.3 User perceived system response time

The system response time as perceived by the user was measured by profiling the code on the PC side, measuring the queuing delay and adding this to the latencies of the various code blocks which were profiled above.

The delay on the PC side for joystick packets is 10ms (which is the sampling rate) + 13 μs to read and send the joystick packets on average.

The mean queuing delay on the FCB side is 10ms and the median delay is $673\mu s$. Combined with the mean latencies on the FCB side this gives a mean response time (as perceived by the user) of 23.4ms. The response time when pressing a keyboard button is lower, since keyboard presses are forwarded to the FCB in the interrupt handler on the PC. Therefore this delay will be approximately 10ms less.

5 PROFILING

Section	Mode	N	$\overline{dur} \pm \sigma_d$ [us]	$\bar{T} \pm \sigma_T$ [us]	dur_{99} [us]	$dur_{99.9}$ [us]
Parse	Safe	22832	20 ± 26	1487 ± 1093	94.7	332
Parse	Full	22525	26 ± 44	1546 ± 1125	250	345
Telemetry	Safe	15	570 ± 62	$1910ms \pm 301ms$	736	752
Telemetry	Full	9	538 ± 33	$1918ms \pm 207ms$	590	590
Control	Safe	7027	10 ± 15	4951 ± 353	58.7	320
Control	Full	7001	63 ± 40	4937 ± 386	175	383
Control RAW	Safe	9250	11 ± 18	2478 ± 246	60	320
Control RAW	Full	9380	113 ± 41	2477 ± 238	220	492
DMP	Safe	6788	1979 ± 145	4949 ± 245	2560	2820
DMP	Full	7090	1983 ± 145	4939 ± 280	2590	2820
MPU RAW - float	Safe	9661	1000 ± 90	2482 ± 251	1410	1590
MPU RAW - float	Full	9970	1001 ± 87	2478 ± 193	1390	1570
MPU RAW - fixed	Safe	18713	1001 ± 99	2478 ± 143	1410	1580
MPU RAW - fixed	Full	18576	998 ± 96	2477 ± 112	1410	1590
Heartbeat	Safe	109	21 ± 7	$324ms \pm 242ms$	48.4	51.7
Heartbeat	Full	99	23 ± 10	$327ms \pm 982$	51.5	75.4
ADC + Baro	Safe	659	189 ± 133	49670 ± 655	444	500
ADC + Baro	Full	707	192 ± 132	49532 ± 2495	437	646
Logging	Safe	492	331 ± 64	4967 ± 270	671	710
Logging	Full	497	319 ± 49	4965 ± 259	509	747

Figure 14: Profiling results. $\overline{dur}$ is the mean duration,$\bar{T}$ is mean period, dur_{99} and $dur_{99.9}$ are the 99th and 99.9th percentile of task durations respectively

6 CONCLUSIONS

In this project stable flight control for a quad-rotor through a wired and wireless medium using a robust communication protocol and carefully tuned controllers and filters, was achieved. Further all safety and general requirements were met with and demonstrated successfully.

6.1 Evaluation

The round-robin architecture we chose in the beginning proved to be sufficient to meet the timing requirements, especially in DMP mode. The limitations of round-robin become clear in RAW mode, when the utilization approaches 100% which eventually lead to important tasks (such as control) being run late. To improve this, we could make a number of changes.

We could define a priority in which tasks need to be run. If a task is ready, a function pointer would be placed in a priority queue. Tasks with higher priority could be run first to ensure that the control loop deadlines are always met.

We could also optimize individual tasks, for example telemetry. The telemetry data is uncompressed at the moment, which means a lot of time is required to send all of the data. Compressing data by sending it at a lower resolution would help to reduce the time spent on this task.

In addition, we could modify the amount of telemetry data sent depending on the amount of time left in the program's main loop. Any excess time could be used to send telemetry data, while if there is no time left we could skip sending telemetry all together.

We could also improve the latency caused by I^2C reads by implementing a non-blocking I^2C driver, such as the one described by Matus Plachy (2013) [10].

The communication protocol and parsing of packets in the queue could also be improved to make it more robust. Packets which need to arrive reliably such as keyboard commands could require ACKs to ensure that they are not missed. The bandwidth could also be reduced by combining multiple packets into a single variable length frame.

6.2 Learning Experience

The project has taught us a lot about the importance of the design phase for embedded systems with timing, performance and memory constraints. It requires a lot more thought than for non-embedded systems which have more memory and a lot more processing power.

A ANDROID APP

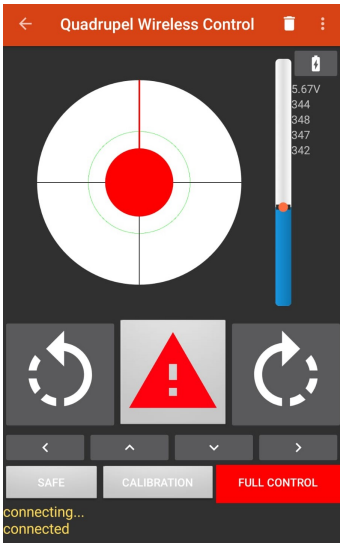

Figure 15: Android app interface

B GUI

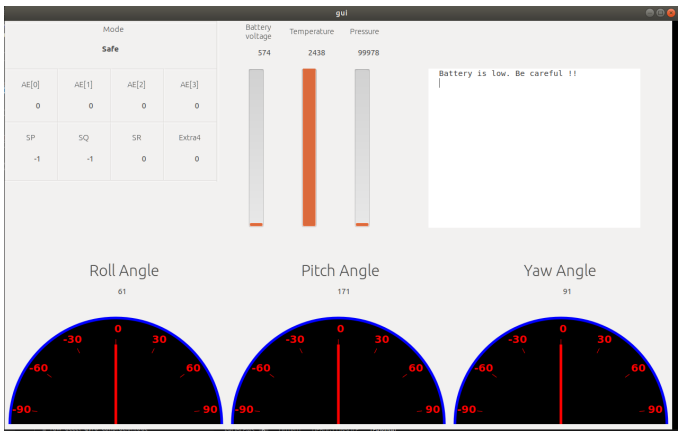

Figure 16: Graphical User Interface

C FEATURED COMPONENT CODE

In order to calculate motor RPM values from L, M, N and Z values, the system of equations in [8, Appendix A] was inverted. The calculation includes taking the square root of 4 numbers. The `sqrt,sqrtf` functions in `math.h` are double/floating point calculations which are very slow on this architecture. Therefore, we implemented an integer version of this calculation using two iterations of the Newton-Raphson method, adapted from [11]. This gives a maximum error of 1 for inputs between 1E6 and 0, and reduces calculation time by $30 \pm 26\mu s$ on average for every square root operation (compared to `sqrtf`) - giving a total time saving of 120us every time the control loop is run.

```
uint32_t xn = 1 << ((32 - clz(square)) >>1);
xn = (xn + (square/xn)) >>1;
xn = (xn + (square/xn)) >>1;
```

Another performance improvement was achieved by replacing the call to `|asin|` in the euler angle calculation in DMP mode (`update_euler_from_quaternions`) with the `atan2` implementation in qfplib. This was done using the following trigonometric identity:

$$arcsin(x) = arctan(\frac{x}{\sqrt{1-x^2}}) \tag{5}$$

When x becomes close to or greater than 1, the regular `asinf` function was used to avoid the singularity which occurs when x >= 1. This should not happen as this implies that the angle is ≥ 45 degrees, which our drone will never achieve during regular flight.

Our implementation in `mpu_wrapper.c:42` saves $503 \pm 74\mu s$ on average for every quaternion to euler angle conversion.

REFERENCES

[1] [n.d.]. Complementary Filter. https://www.sciencedirect.com/topics/computer-science/complementary-filter.

[2] [n.d.]. Complementary filter - My IMU estimation experience. https://sites.google.com/site/myimuestimationexperience/filters/complementary-filter

[3] [n.d.]. (PDF) System Identification Toolbox for use with MATLAB. https://www.researchgate.net/publication/37405937_System_Identification_Toolbox_for_use_with_MATLAB

[4] [n.d.]. TKJ Electronics. http://blog.tkjelectronics.dk/2012/09/a-practical-approach-to-kalman-filter-and-how-to-implement-it/

[5] Petteri Aimonen. 2011. Libfixmath. https://github.com/PetteriAimonen/libfixmath/.

[6] FreeRTOS. 2020. STM32F051 ARM Cortex-M0 FreeRTOS. https://www.freertos.org/FreeRTOS-for-STM32F051-Cortex-M0-IAR.html. Online; accessed 1 Nov 2020.

[7] Kai Morich. [n.d.]. SimpleBluetoothLeTerminal. https://github.com/kai-morich/SimpleBluetoothLeTerminal. Online; accessed 28 Sept 2020.

[8] Arjan J.C. van Gemund Koen Langendoen. [n.d.]. CS4140 Lab Assignment 2018-2019.

[9] Klas Löfstedt. 2018. Altitude estimation using accelerometer barometer. http://www.klaslofstedt.se/altitude-estimation-using-accelerometer-barometer/.

[10] Matus Plachy. 2013. I2C Non Blocking Communication. https://www.nxp.com/docs/en/application-note/AN4803.pdf.

[11] User: Kde. [n.d.]. Stackoverflow - Efficient Integer Square Root. https://stackoverflow.com/a/51585204. Online; accessed 25 Sept 2020.

[12] Object Tracking: 2-D Object Tracking using Kalman Filter in Python says:, Rian says:, Ryan says:, Rahmadsadli Says:, Manish Kumar says:, and JayMnM says:. 2020. Object Tracking: Simple Implementation of Kalman Filter in Python. https://machinelearningspace.com/object-tracking-python/

[13] Arjan J.C. van Gemund. [n.d.]. QR Controller Theory.