# Embedded Software Laboratory Final Report

**Team 15**

***Prithvish Nembhani (5352576) ; Ritik Agarwal (5546419) ;***

***Ghanishtha Bhatti (5457025) ; Berkay Yazıcıoğlu (5564395)***

**Abstract:** This project aims to build flight control software for the operation of a Quadrupel (QR) drone from scratch. The scope of the project spans from enabling simple joystick-based manual control to implementing automatic motion stabilisation through cascaded P control. The developed software acknowledges constraints on safe states, memory, and power of the QR flight control board (FCB) by implementing several failsafe mechanisms as well as software workarounds for hardware inadequacies. The software was rigorously tested on the physical system and tuned to optimise for inconsistencies between assumptions and hardware performance. The final version of the project was able to achieve the set control goals to a reasonable extent, though further design iterations could have improved the robustness of the system and ironed out the minor flaws that were observed during its operation.

**Keywords:** Quadrupel Drone, Embedded Software Development, Embedded Control System.

## 1 Introduction

Unmanned vehicles have been one of the most commonly used robots in the industry and are being used for many applications such as near earth surveillance, videography, distance photography, agriculture, delivery and much more. Since, UAVs have become one of the most popular embedded systems around us, understanding the practicals makes it important along with being fun and interesting. The project aims to build the embedded software from a baseline code in C and achieve certain stages of the UAV operational in a duration of 8 weeks. The project integrates expertise from several disciplines such as communication, mechanics, control theory, sensor and actuator electronics, signal processing, computer architecture and software engineering. The prime focus of the course will be on building a robust embedded software with the freedom to make choices on the software architecture, communication protocol, control system etc. Where drones remain a huge area of research in the field of controls, the project does focus on modelling drones and control theoretical but rather on the application side of control. The further sections describe the details of choice of software architecture, communication, implementation details, and the experimental results obtained.

## 2 Software Architecture

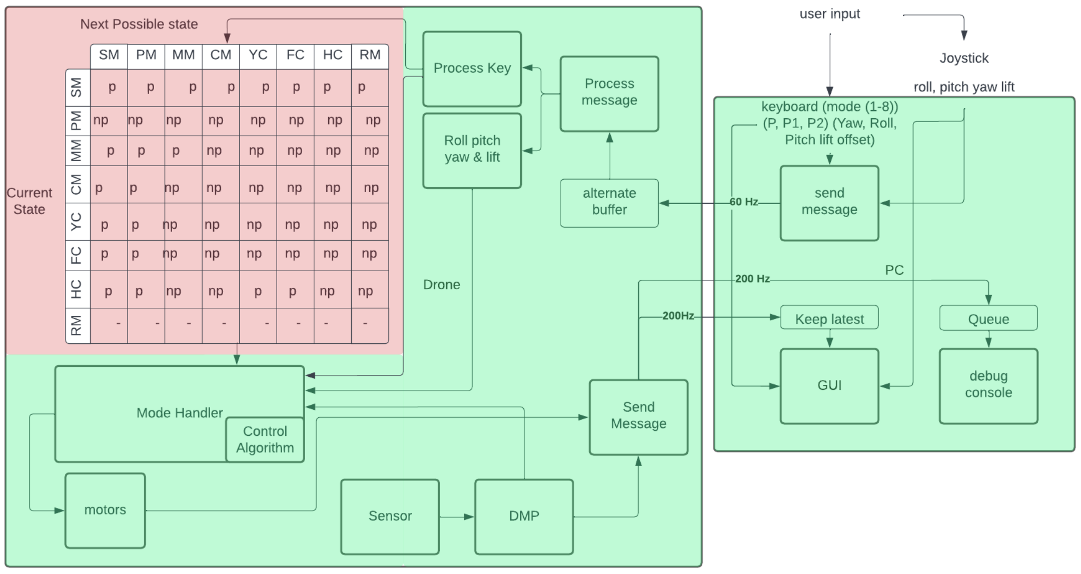
The software architecture can be thought of as a round-robin with interrupts. The main loop would service the tasks like receiving data from the drone, updating motor, getting sensor values, running the state machine would be serviced in a round-robin manner whereas messages from the PC with mode changes would be treated as events and served as interrupts to the system. The green section in the figure 1 represents the section of the code which will be executed in a round-robin fashion and the red section in the red section on interrupts from the user for mode change, usb disconnects, communication failure etc. The state machine is implemented in a manner that if the execution of the mode is not finished in the given time duration before the deadline, it would be interrupted by the events. The events could be going to panic mode or any other mode. The 2.1 and 2.2 subsections describe the state machine for the mode change and communication from the PC to drone and drone to PC in a detailed manner. **??** gives details on the timing of main sections of the code.

### 2.1 Main Modal State Machine

The State Machine was designed as a two-dimensional lookup table with the allowed mode transitions marked in the intersection of the row-column mode pairs. The contents of this table are the current mode in which represents the current state of the system and the other thing is the event received from the user. The event can be thought of as a transition which takes the system from one mode to another. If the system is in some mode and it receives a particular event, then based on the table it is checked if the system is allowed to make the transition to that mode or not. In the 1 the permissible transitions are show by a **p** and non-permissible transitions are shown by a **np** The mode changes are handled by the keyboard inputs, which behaves as hardware interrupts.

The different events or transitions which were used for this state machine was the same as the number of modes being implemented, **mode 1-8**. One additional event was the Null event, which enabled the current handler to be called continuously when no mode change interrupts were being given to the system. In cases where a certain, transition was not allowed from the current mode, that transition was treated as a null event and the system remained in the same mode. The null event can be thought of as a self loop of the state.

Six modes were developed for the Drone and the number keys on the keyboard were mapped to enable a mode change. Each mode had a specific mode handler defined for it which contained the logic for performing the tasks of that mode. Each mode had its specific deadline set in which it was expected to finish its processing, and this is how the scheduling of different tasks was handled in the system. Depending on the requirements, different modes had varying deadlines. The complete software architecture of the QR flight control system is illustrated in Figure 1.



| | SM | PM | MM | CM | YC | FC | HC | RM |
|---|---|---|---|---|---|---|---|---|
| SM | p | p | p | p | p | p | p | p |
| PM | np | np | np | np | np | np | np | np |
| MM | p | p | p | np | np | np | np | np |
| CM | p | p | np | np | np | np | np | np |
| YC | p | p | np | np | np | np | np | np |
| FC | p | p | np | np | np | np | np | np |
| HC | p | p | np | np | p | p | np | np |
| RM | - | - | - | - | - | - | - | - |

Figure 1: Software Architecture Block Diagram

## 2.2 Communication Protocol

The communication between the PC and drone can be majorly divided into 2 parts. The communication from Drone to PC and from PC to Drone.

### 2.2.1 Drone to PC

The Drone to PC communication majorly involved logging data from different sensor values and motor values at any given point of time. While the communication could be kept a bit unreliable, we had to make sure that due to the communication from Drone to PC the PC to Drone communication is not affected which creates delay in operation in the end resulting in increase in responsiveness. To assure the same, the communication from Drone to PC was telemetry based and handled on a different thread on the PC side. Since our implementation had a GUI implemented as well, all essential motor values and sensor readings were segregated from the debug messages via delimiter **GUI** and **debug**. Values extracted from the **GUI** section were to be used in the real-time GUI on the PC side. Values extracted from the **debug** were to be used to print messages and debug information sent from the drone side. The data frame send from the Drone to PC is a variable length message due to the debugging mechanism implemented on the Drone side Hence, the fields are as follows 1. On the PC side, the buffer reflects the latest values for the GUi whereas all the values after the debug are appended to the buffer and printed on the command line. While the drone is not sending any message, the buffer drops empty packets from the wire and does not process it.

### 2.2.2 PC to Drone

The PC to drone communication consisted of majorly combining the Joystick (Extreme 3D Pro) control with the keyboard keystrokes and encapsulating in a message and sending it to the drone. The main focus on this side of communication was making sure there is no corruption or loss in package and the communication is reliable. To assure reliable communication, on the drone side, the size of the message was checked for the exact number of bytes. Apart from this, an alternating buffer was implemented which made sure that at any point in time there would not be a situation when 2 consecutive messages are received and the drone is not ready to process them. The latest of the joystick and keystrokes were to be used for the motor calculations. The whole communication was limited to 16ms from the PC side so that the Drone is not always processing messages, and it does not perform any other tasks such as control loop calculations. The data frame send from the PC to Drone is a fixed length message and the fields are as follows 2. The limitation of the communication protocol was that it would not check for corruption in the packet sent. But in general there was no corruption in package noticed which would disrupt the operation of the flight, we kept the CRC check as a part of future implementation. On the RS232 UART link, the packets would be sent character by character and bitwise OR and masking operations were used to combine fields which would be more than 1 byte.

The valid keystrokes include keystrokes for panic, changing the gains for P, P1 and P2 and changing the offsets for yaw, lift, roll and pitch.

| | |
|---|---|
| delimiter "gui" | 3 bytes |
| mode | 1 byte |
| yaw offset | 2 bytes |
| roll offset | 2 bytes |
| pitch offset | 2 bytes |
| lift offset | 2 bytes |
| motor 0-3 | 8 bytes |
| phi | 2 bytes |
| theta | 2 bytes |
| psi | 2 bytes |
| P | 2 bytes |
| P1 | 2 bytes |
| P2 | 2 bytes |
| battery | 2 bytes |
| temperature | 2 bytes |
| pressure | 2 bytes |
| delimiter "debug" | 5 bytes |
| debug info | n bytes |

Table 1: Drone to PC Packet Format

| | |
|---|---|
| start | 1 byte |
| key stroke | 1 byte |
| yaw value | 2 bytes |
| roll value | 2 bytes |
| pitch value | 2 bytes |
| lift value | 2 bytes |
| end | 1 byte |

Table 2: PC to Drone Packet Format

## 2.3 Scheduling and Profiling

The major profiling had to be on the Drone side main loop to make sure the drone is responsive to the changes of the joystick. To make sure the same is achieved, the code was profiled to the following values.

| Task in the main loop | Time |
|---|---|
| Receive data | 16 ms |
| Send data to PC | 2 ms |
| Log sensor data | $<$ 1 ms |
| State machine(yaw, full, manual, safe) | 1 ms |
| state machine (panic) | 200 ms |

Table 3: Profiling Values

To make sure every mode runs within the limits of execution to maintain responsiveness, every mode was implemented with a time period and deadline which could be altered based on the instructions/code it would execute in the given mode. In our implementation, the time period were set to 1 ms except panic mode. The time period for the panic mode is set to 200 ms. To make sure the GUI was updated to the latest values sent from the drone, profiling was done for sending data and GUI update on the PC side as well.

## 3 Implementation

### 3.1 Universal Control Functions

While specific functionalities are programmed for the individual modes, several general features are developed and used extensively by every mode. These include the following major utilities:

- *Set and Reset Motors*: these functions allow values from the software to be sent to the motors as speed commands. The set motors function takes as arguments the speed values to set the four motors to, as well as the minimum and maximum allowed RPM for a particular mode. It then performs validation on the user/algorithm specified input speeds to the motors to verify whether they are within the accepted RPM range. The lower bound of this range is the minimum RPM of 175 at which the motors do not stall. The upper bound is a safe flight value of 600 RPM for the full control mode, and 450 RPM for the remaining modes. If the input values fail the validation, they are instead saturated at these lower and upper bounds. This way we ensure that regardless of the output of the control algorithm or joystick mapping, we never explore any unsafe states (stalling or incredibly high speed) on the physical QR. The reset motors function is primarily used when stopping the QR completely, as it sets the motor speeds to zero.
- *Mapping Limits*: this function is used for scaling input range to an output range, and is used extensively to maintain the mappings in all the flight modes. It takes as arguments the lower and upper bounds of the input range as well as the desired output range. It also takes the raw input value that we want to translate from the input to the output range. The joystick gives values ranging from $-2^{15}$ to $2^{15}$ for each QR axis (roll, pitch, yaw, lift), therefore, the input range to the function was consistent in all modes. The output range was customized based on the derivation of the mapping equation and our required physical constraints for each mode.
- *Sensor Bias Deduction*: this function is used to translate every incoming raw sensor reading by the offsets computed by the system calibration (subsubsection 3.2.4). The translated sensor readings are then utilized by the controllers developed in the control modes (subsubsection 3.2.5 and subsubsection 3.2.6).

### 3.2 Modes and their Operation

#### 3.2.1 Safe Mode

The safe mode is used for standby/idling the QR. It has been programmed to ignore any input from the joystick or the keyboard except if it entails a change in mode. Furthermore, some safety conditions have been enforced through the state machine that prevent the operator from leaving this mode in unsafe conditions. The condition enforces that the joystick is in the neutral position, indicating zero input to the motors. For most of the implemented modes, changing between them may only take place through the safe mode. However, entering the safe mode directly gives zero values to the motor speeds, so it is preferred to leave a flight mode through the panic mode (unless one wishes to witness the QR falling out of the air like a brick). The safe mode is also the mode at initialization of the QR.

#### 3.2.2 Panic Mode

There were several scenarios under which the drone should go into the panic mode in order to ensure safety. There were keyboard and joystick inputs to take the drone into panic mode and also events like loss of communication from the PC side which should cause the drone to enter the panic mode. For the implementation of the panic mode, one of the ideas proposed was to set all the motor values to a value which corresponds to the current highest motor RPM among all the motors. This solution was proposed in order to stabilize the drone by giving all the motors an equal RPM value and then decrease the motor values gradually to zero. This solution seemed to work well for the manual mode and the yaw control mode, as there was normally no big difference in the motor RPM's under these conditions.

While checking the implementation of the full-control mode, the difference in rotor speed's of the different motors was much more under certain scenarios. In case of a panic situation under such a condition, there was a significant revving up of the motor RPM for motor's rotating at lower speed. Though this was supposed to happen and was thought of as the best solution to stabilize the drone before landing, the fact that some of the motors revved up to match the speed of the fastest motor did not receive positive feedback, so the approach had to be reconsidered. In the final version of this mode, we modified the function to first level all the motors to the speed of the slowest motor and then gradually slow down uniformly.

### 3.2.3 Manual Mode

To implement joystick based control of the quadcopter, this mode was used extensively. Through the duration of the project, we experimented with two different methods of translating the raw joystick values to motor mappings. Originally, we imagined the most intuitive method of achieving manual control was to use a direct linear mapping of the joystick values for roll, pitch, yaw, and lift onto the motor values that they represented. This method worked well in practice as well, because in the manual mode there is no sensor-based control action that will be affected by the physical non-linearity of the system architecture. Therefore, linear mappings hold their ground well here.

However, we realized that the same intuition would not hold with the control modes where the angular or rate error affects the moments and the mapping to the motors can no longer be assumed to be linear. In order to maintain consistency in the modes and strengthen the mathematical foundation of the implementation, we switched to the mapping referenced in the assignment manual by reverse mapping the moments (M,L,N) onto the motor RPM values (m0-m4). Furthermore, this method posed one additional challenge, as it required the square root operation to compute the mapping. As the FCB does not possess any floating point unit, we instead had to utilize a software floating point function. Thankfully, there are developers who have shared open source code for this exact purpose [1]. Using this function, the mapping to the motors synchronously transformed joystick input for roll, pitch, yaw, and lift into specific motor speeds. This mapping had to be fine-tuned through a scaling factor to ensure that the user input through the joystick gave a commensurately aggressive response to QR motor speeds. The same basic mapping was used in both the control modes, with the exception of the scaling factor, which had to be tweaked to allow us to visible notice the impact of the controllers.

### 3.2.4 Calibration Mode

When using the sensors on board the QR, it was of the utmost importance that the neutral position did not give any sensor offsets, which would give anomalous control actions in the control modes. To nullify these offsets our implementation of calibration mode finds the sensor offsets for roll and pitch angle as well as yaw rate. This is done by storing the mean of the sensor readings in the neutral position over 5 seconds, and storing this mean as part of the drone message. As we have these values being sent with the messages until they are overwritten by reentering the calibration mode, the offsets are simply subtracted from the raw sensor readings before being used. While this approach worked considerably well for the roll and pitch angle offsets, they could not account for drift (increasing error in sensor readings) that was observed in yaw rate. A possible solution to this could be to find the approximate gradient by which the values increase over time and subtract a time-varying variable from the raw readings. However, since the accuracy of the yaw rate error is not as significant as roll and pitch angles, we decided to stick to the basic offset deduction approach.

### 3.2.5 Yaw Control Mode

In this mode the first instance of applying control action based on sensor readings was implemented. The yaw rate, was extracted from the Digital Motion Processor (DMP) unit onboard the QR. The mapping of rate onto moment/torque has a phase lag of $90^\circ$, which can be bridged through a single controller. To maintain the most computationally simple control design, a proportional gain (P) controller was used. The error for computing control action was found by subtracting the sensor reading from the reference rate taken from the joystick input. A tunable proportional gain amplifies the error between actual and required trajectories, and generates the change in yaw moment (N). Correspondingly, the mapping discussed in subsubsection 3.2.3 was used to translate this control action onto the four motor speeds.

One discernible caveat of the original implementation was that using the same scaling as the manual mode gave us very poor resolution on the control gain. A gain of 1 gave an almost unstably aggressive response to error. To circumvent this issue, we introduced a separate scaling factor for the yaw moment in the motor mapping. Some experimentation with this scaling factor, gave us reasonable tuning resolution for the proportional gain. Another challenge we faced was the growing drift of the yaw rate. Due to this over time an increasing error was being produced even for the stationary QR. The way to neutralize this problem (atleast to levels that were undetectable visibly or tangibly) was to choose a balanced value for P that provided good aggressive response, but did not amplify the drift error too much.

### 3.2.6 Full Control Mode

The full-control mode has been implemented for the control of pitch and roll along with the yaw control. A simple extension of the yaw mode control wouldn't be sufficient for controlling the pitch and roll as that will require mapping a angle set point to a torque set point, which will give rise to a 180° phase lag which would lead to an unstable controller. To deal with this a cascaded P controller was used where one of the P controller maps the joystick angle set point to the rate set point and the other P controller maps the rate set point to the torque set point which is directly used to alter the Drone's engine RPMs. The cascaded P controller behaves a PD controller as it is able to neutralize one of the integrator and hence gives a stable control under the condition P2≥4P1.

The joystick and keyboard controls was the same as in yaw control mode and the deadline for this mode was set at 1ms. In addition to the 'P' gain for the yaw control, two new gains 'P1' & 'P2' were introduced to be the gains of the cascaded P controller for pitch and roll control.

For the implementation of the full control mode the upper limit of the drone's rotors was raised to 650 RPM so that the effect of the controllers could be observed and also this enables to test out near hover conditions on the drone. For this the relevant mappings of the "Z" was changed. For pitch and roll control, the maximum permissible angle of tilt input was set at 60° as it was thought that in case of tilt angle higher than this would make the drone flip and lead to a crash. To enable this the relevant mappings of the joystick inputs to 'M' & 'L' were modified, It was noticed that the range of the roll angle sensor was half the range of the pitch angle sensor and therefore there was a difference in the mapping for 'M' and 'L'.

One flaw of the original implementation of this mode was that due to the use of mismatched sizes in the variables, the moments associated with the roll and pitch sometimes experienced overflow when the angle of tilt was very high. This overflow meant that at the direction of the control action was reversed beyond a certain threshold, which caused a positive feedback loop rather than a negative feedback loop. After some modification of the variable sizes to be compliant to the constraints, the implementation was free of this problem.

Due to the large amount of offsets in the theta and phi values coming from the gyroscope the drone was unstable even when being calibrated and had some vibrations when it was in a neutral position. To counter this a small range of values from roll and pitch sensor was ignored as noise and to eliminate the vibrations caused by the offset.

### 3.2.7 Height Control Mode

The height control mode can be seen as an extension of the control modes, rather than its own standalone mode. It follows from this intuition that the scheduling and variables are carried forward from whichever mode we originated from before going to height control mode. This means that we are still operating with the same behaviour as the original mode, but with the added utility of maintaining the height.

The objective is to maintain a flight altitude setpoint which is set based on the last value of the altitude before we switched into height control mode. The altitude is sensed on the QR through the onboard barometer values. A proportional gain controller in this mode tries to counteract changes in the barometer readings such that the altitude of the drone is maintained at the setpoint.

A key difference between this controller and the controllers in subsubsection 3.2.6 and subsubsection 3.2.5 is that the setpoint cannot be changed by the user once we enter the mode. It follows from this that toggling the throttle switch on the joystick (meaning the user wants to manually vary the height and not remain stabilised at the previous setpoint) automatically disables the height control mode. When this happens the QR reverts back to the previous control mode.

Though we were able to design the handler sufficiently for this mode, we were not able to test it rigorously due to insufficient time. In its current implementation, the mode does not seem to reflect an aggressive enough control to verify tangibly.

### 3.3 Graphical User Interface

The Graphical User interface was real-time clickable GUI made in python using tkinter libraries. The GUI had fields for current motor values, offset values for yaw, pitch roll, lift, Euler angles, gain values for P, P1 and P2, buttons to start and stop the communication with the drone. To maintain the responsiveness of the GUI the backend of the GUI takes in only the latest values from the messages sent from the drone. In the top right corner, there is a real time plot of the drone, which would simulate the drone in real-time in the absence of a drone. The GUI made it very convenient for us in the development process in designing a robust algorithm by simulating the position of the drone in a plot and giving the real-time motor values. 2 gives a graphical implementation of our GUI.

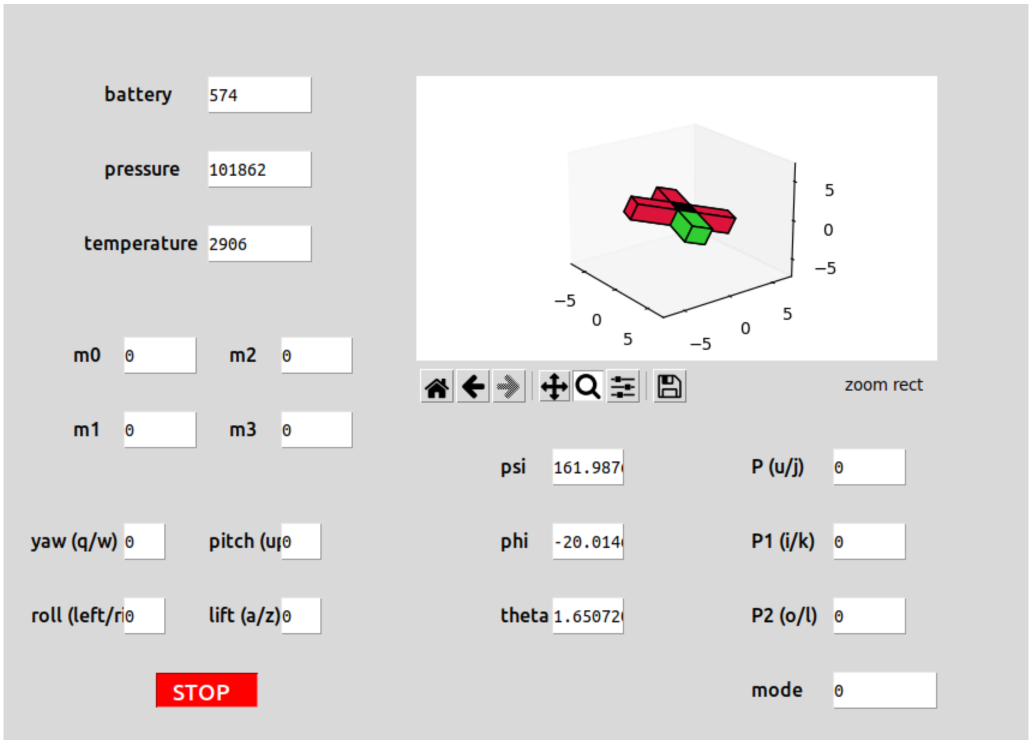

battery 574
pressure 101862
temperature 2906
m0 0
m2 0
m1 0
m3 0
yaw (q/w) 0
pitch (u| 0
roll (left/ri 0
lift (a/z) 0
psi 161.987
phi -20.014
theta 1.65072
P (u/j) 0
P1 (i/k) 0
P2 (o/l) 0
mode 0

Figure 2: Graphical User Interface of the Developed QR Flight System

## 4 Experimental Results

The capabilities of our project have evolved over subsequent iterations of development. A modular development approach allowed us to test each mode/function separately making the debugging process quite intuitive. We ensured that the function/mode was performing sufficiently on the standalone FCB before testing it on the powered up drone to prevent avoidable damage. In the ideal case, this would enforce the "first-time-right" principle, but as is often the case with hardware, there were often corner cases on the code that were only visible during actual operation. Subsequent troubleshooting and profiling, allowed us to iron out these issues to a large extent. In the final design iteration, our project was able to demonstrate the following abilities.

- Safety Features:
  - Prevention from leaving safe mode if joystick is not in neutral position.
  - Automatically enters panic mode when connection to PC is severed or communication is not received for some time.
  - Responsive mode transition to panic mode when panic button is pressed.

- Manual Mode Operation:
  - Quick response to joystick commands.
  - Throttle mapping varies directly from 0 to 175 RPM (minimum no stall speed) to prevent motors stalling.
  - Sufficient mapping of raw joystick values to tangibly verify the effect of varying a specific QR axis on the joystick.
  - Trimming of roll, pitch, yaw can be performed spontaneously by the user during operation.
- Control Modes Operation:
  - Quick response to joystick angle and rate setpoints.
  - Sufficiently robust to base level sensor errors.
  - Produces a tangible and visible control action after the gains are tuned reasonably.
  - Is able to track the reference with minimal observable error.
  - Is able to reasonably counteract disturbances to QR in any configuration.
- Visualisation Features:
  - Real-time sensor data and motor speeds can be observed via the GUI
  - The matplotlib animation updates in real-time with readings from drone sensors.
  - Shows current values for the control gains and is updated in real-time when they are changed.
  - Calibration on the drone can be visually verified through the animation and sensor readings on the GUI.

The final size of the C code that was uploaded onto the FCB is delineated in 4.

| code subsection | bytes |
|---|---|
| text | 49156 |
| data | 180 |
| bss (block started by symbol) | 2644 |
| dec (total) | 51980 |

Table 4: C Code Size Uploaded to FCB

To get a better understanding of how the software performs after the profiling, multiple iterations of the loop were left for execution by changing modes of the code at random and testing how the code performs. The timing profile of the state machine can be seen belowFigure 3 respectively.

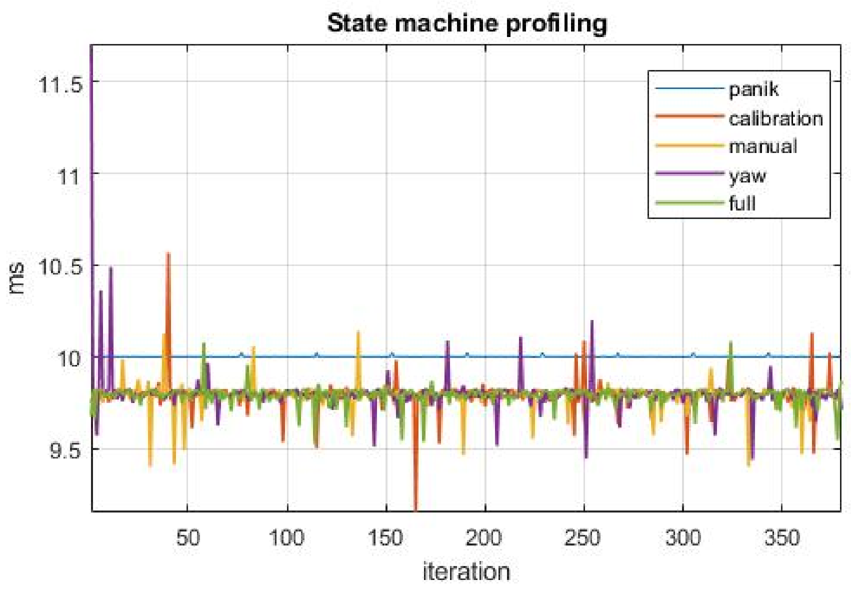


Figure 3: State Machine Timing Profile

After profiling the communication between the PC and drone and viz versa, multiple iteration of the code were executed on the flight controller and a rough estimate of communication latency was estimated. Figure 4 shows the communication latency for Drone to PC and PC to Drone communication.

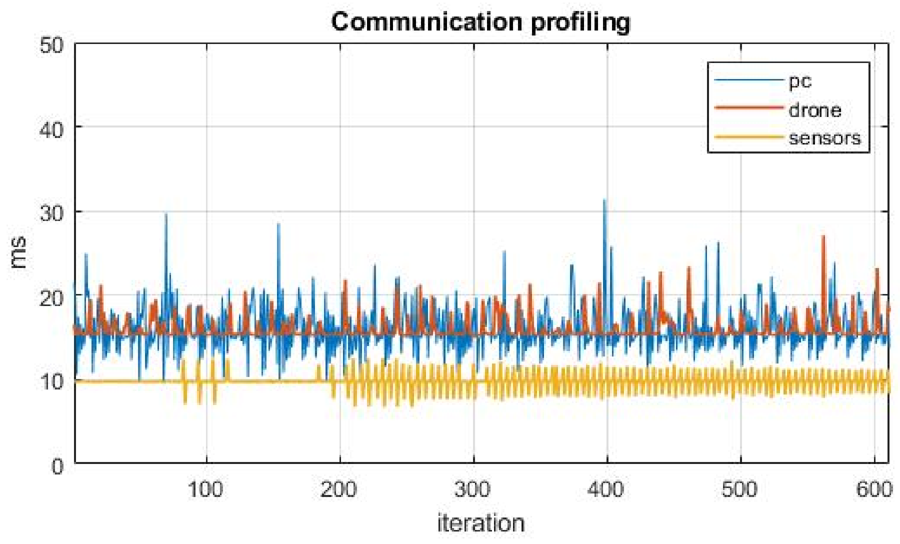


Figure 4: Communication Timing Profile

## 5 Conclusion

We believe the design of our project was well thought out and robust. The communication protocol and the safety features were well thought and discussed in great detail by all teammates. There were several implementations we tried out and some of them didn't receive the feedback we expected or just behaved in a way which was different than initially thought of. These experiences and feedback made us realise which design choices work and which do not.

Every team member was involved when discussing the implementation of the different modes of the state machine and while discussing the state machine itself. It took us some time to figure out the communication protocol but once we overcame that we were able to get to the manual mode quickly. After the midterm feedback we realised that although our manual mode implementation worked well, the logic behind it was not correct and that meant we had to rethink our implementation for both the manual mode and the yaw mode. This proved to be a setback in our progress and hence we were not able to go beyond the implementation of the full-control mode.

Overall this entire project has been a steep learning curve and an opportunity to get hands on experience in dealing with embedded software and control system design combined together. There were several areas in which we could have and would have liked to improve but we ran out of time. One such area of improvement was the delay between the input at the joystick and observing the output on the drone. Though the drone was really sensitive to changes in the joystick position it wasn't as fast to respond to those changes and this caused a visible delay between the input and the output. We had several ideas for the implementation of the raw mode and the height control mode but we got stuck in the implementation and fine-tuning of the full control mode and hence didn't have enough time to test those implementations.

## References

[1] "Mozzi: cogl_sqrti.h Source File," 2020. Available at https://sensorium.github.io/Mozzi/doc/html/cogl__sqrti_8h_source.html, version 1.8.14.