Embedded Systems Lab

Group 4
Li Ou Hu (5236541), Huixuan Wu (5885310),
Chenruiyuan Yu (5930162), Dielof van Loon (5346894)

May 28, 2024

Abstract

For the embedded systems lab, software was written for flying a drone with an embedded system, which is controlled through a host PC. The host PC is interfaced through a joystick, keyboard and a GUI. The safety features, manual mode, and full control mode have been demonstrated to be fully working. On top of these features, the raw control and height control modes were partially implemented.

1 Introduction

The embedded systems lab is a lab where the host PC controls a drone through the host PC as shown in 1. This drone has 4 motors, which can be controlled individually. The goal is to implement system control functionalities to the drone to keep the drone flying in the air with sufficient stability.

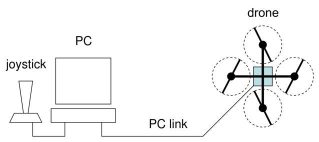


Figure 1: System architecture[1]

The project consists of several main parts. First of all, a serial communication protocol is established through the physical PC link. Then, the drone motors are controlled directly through the joystick by mapping values as described for the manual mode. When the drone can be properly controlled through the joystick, the calibration mode is implemented together with the yaw control mode, to implement the first control system for the yaw-angle of the drone. This control system is then extended to both the pitch and roll, in order to form the full-control mode. Then, the sensor values are directly read from the drone instead of the DMP, in order to obtain a finer granularity in the control of the drone. At last, an implementation of the height mode has been designed, which keeps the drone flying at a specific height.

2 Architecture

This section will describe the software architecture of the complete system. Essentially, the system exists of two components, namely the PC and the drone, as shown in Figure 1. The PC and drone each run their Rust code individually, which can communicate with each other through the PC link.

Figure 2 shows the software architecture of the entire design, with the runner code architecture detailed out. The runner code is run on the PC side, which serves as an interface for the user. This runner code takes in the joystick and keyboard inputs from the user. Additionally, the runner code has an implemented GUI, which can be used to display data from the drone and to send information to the drone. This GUI is implemented in Python with the PyQt library and interacts with the Rust code by sending data through a `output.txt` file. All of these functionalities are performed in each thread, and within a main loop, the actions of each thread are aggregated. Then,

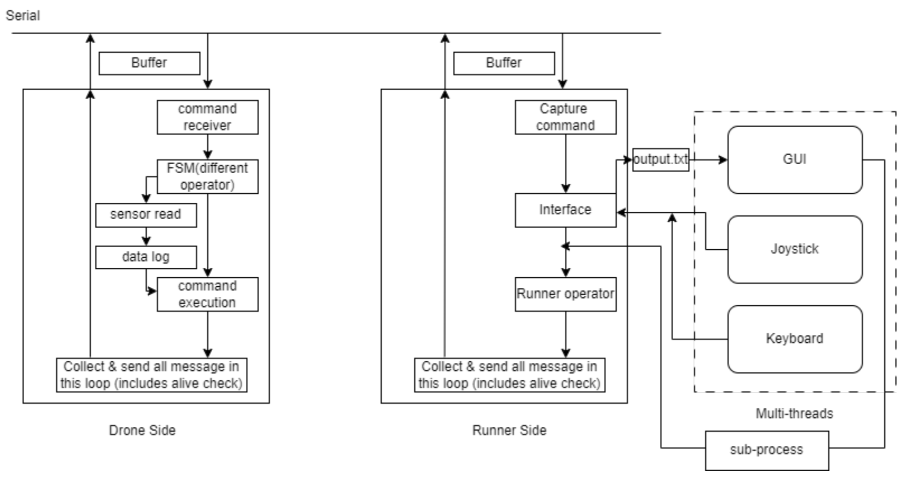


Figure 2: Software architecture, with emphasis on the runner side.

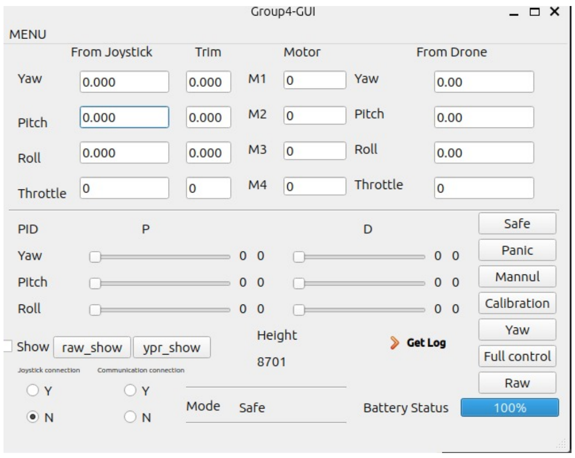
information is sent and received to and from the drone through a serial communication protocol implemented at the PC link.

The drone runs an embedded Rust code, which at a high level is represented by a finite-state machine (FSM). This FSM is operated with in the embedded Rust code through an infinite loop. At the end of every iteration in the infinite loop the code will wait until a tick happens, of which its frequency can be set. In each iteration, the drone will try to read bytes from the serial communication buffer and deserialize the buffer in order to obtain a message. If a message has been deserialized, corresponding values in the drone are changed based on the current mode. Then, the drone will perform its operation based on its current mode and variables. This operation is done even if no messages have been received. Each state in the FSM, determines the behavior of this drone operation. These implemented modes are: Safe mode, panic mode, calibration mode, yaw-controlled mode, full-control mode and raw control mode. Of each mode, its implemented operations will be described in the following section.

3 Implementation

3.1 Runner code

The runner code on the PC side is responsible for the joystick and keyboard input, receiving and sending commands to the drone through the PC link, and showing a graphical user interface (GUI) as seen in Figure 3 to the user. This GUI displays the joystick input values and the current state of the drone. The GUI is made by QT designer and is converted to a Python script. It has two changes based on this. At first, the GUI and rust runner work in parallel on the PC. Then, the GUI reads the `output.txt` file at a certain frequency through a thread, which shares the real-time information of the drone and joystick with the runner and displays it at the relevant location. Next, the GUI uses a child process to send commands to the runner, which in turn sends them to the drone to enable the GUI to control the drone. Furthermore, we have implemented the keyboard and the joystick to the runner by using two Rust dependencies, `termion` and `gilrs`, respectively. The GUI (in the Python script named `demo.py`), requires an environment with `pqyt5` and `Python3` to run.


Group4-GUI

MENU

	From Joystick	Trim	Motor			From Drone
Yaw	0.000	0.000	M1	0	Yaw	0.00
Pitch	0.000	0.000	M2	0	Pitch	0.00
Roll	0.000	0.000	M3	0	Roll	0.00
Throttle	0	0	M4	0	Throttle	0

PID	P		D	
Yaw	0	0	0	0
Pitch	0	0	0	0
Roll	0	0	0	0

Height 8701

Joystick connection: Y [✓ N]

Communication connection: Y N

Mode Safe

Battery Status 100%

Figure 3: GUI made with Qt Python

3.2 Serial communication

The serial communication that was implemented has messages of varying lengths. These messages are serialized commands, which the drone and host PC can interpret, and retrieve its contained data (if any) by deserializing the message. Both the drone and the runner side transmit information by sending and receiving buffers, as shown in Figure 5. The serialization and deserialization of messages is through the postcard and serde dependency, which hardly uses the bytes 0xFE and 0xFF. Therefore, we set the start byte and end byte of the protocol as 0xFE and 0xFF respectively. In Figure 4, the structure of the communication protocol is shown. The beginning of these messages is indicated by a start and stop byte. In order to maintain the integrity of the messages, and synchronization between the host PC and drone, a cyclic-redundancy check (CRC) has been implemented, along with a byte indicating the length of the message. This CRC and checksum, both consist of 2 bytes. The CRC is completed by the dependency of crc_any via a special polynomial. In the case that a byte within the serialized message, it might occur that the message is terminated abruptly. For this, the byte which indicates the length of the message can be used to check if this is the case or not. With this implementation, variable-length messages are used to reliably send information between the host PC and the drone.

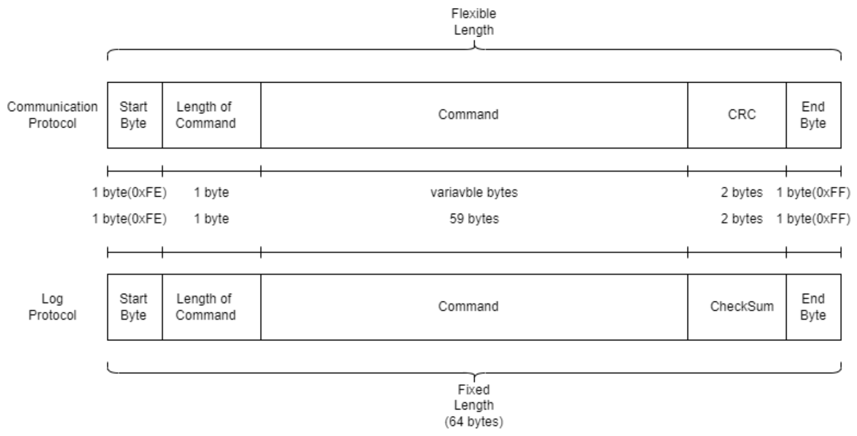


Figure 4: Structure of the communication and log protocols

3.3 Drone code

Similar to the runner code, the drone code operates in a infinite loop. The execution of each loop iteration is limited by the tick frequency, which has been set to 200 Hz. This tick frequency must not be faster than the time required to execute an iteration. When the tick frequency too fast, the execution time of a loop iteration is unknown. As several timing mechanisms are dependent on this tick frequency, these mechanisms will break in this case. These mechanisms are for example determining the time duration since a byte has been received through the serial communication, or to simply blink a led every second.

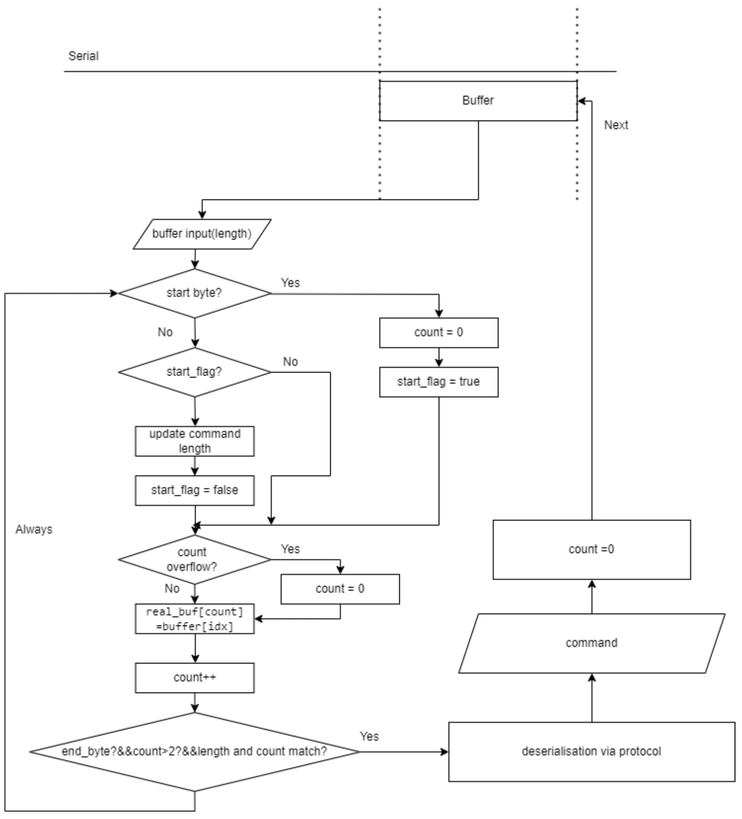


Figure 5: Work process of communication

Each loop iteration, several functionalities are executed sequentially. First, the serial communication is read, and a message is deserialized (if any). If a message has been successfully deserialized, the drone processes this message and changes its own internal state if the drone accepts this message. For example, the drone can only go to the panic mode from a mode where the drone motors are operating. When the runner code transmits a message to switch to for example yaw-controlled mode to full-control mode, this message is rejected, and the drone does not switch modes. Additionally, any active joystick input will cause the drone to reject messages to change modes.

Then, after processing the received message, the drone will perform its operation based on the current mode. The mode of the drone, together with the current state of its parameters, are kept track of through an implementation similar to a finite-state machine.

3.4 Safe mode

In the safe mode, the motors are not spinning, and thus the drone should not be moving at all. From this mode, the user can access other modes. Any other commands coming from the runner code are refused. Additionally, if the throttle of the joystick is not 0, any command to enter another mode is refused.

3.5 Panic mode

The panic mode is used to immediately abort the operation of the drone as safe as possible. When the drone enters the panic mode, the motor speeds equalize and gradually decrease to 0 over 1 second. Then, when the motor speeds are 0, the drone enters the safe mode.

This panic mode can be entered manually by the user through a joystick or keyboard input. Furthermore, the drone automatically enters panic mode during uncontrolled behaviour by the following functionalities:

1. Detect when the battery level drops too low.
2. Detect when the PC link is disconnected / inactive. This is done by issuing a keepalive command if there has been no communication in a predefined interval. When the drone has not received any bytes in this interval, the drone will enter the panic mode.

3. Detect when the joystick has disconnected. The runner code will issue a panic mode command whenever a joystick disconnect event has occurred.

The keepalive command is implemented by having the host PC send a message, if no messages have been sent within the last 200 ms. This way, when the host PC is actively sending messages, the keepalive command does not send any additional messages.

3.6 Manual mode

Manual mode is used to directly test the connection from the joystick to the drone motors. Manual mode maps the joystick state directly to the motors. This means that adding pitch on the joystick should quadratically increase the difference in RPM's of the front and back motor. The quadratic relation is due to the proposed mapping of motor values to force and moment generated by the motors, where Z is downward force, L is the roll moment, M is the pitch moment and N is the yaw moment.

$$\begin{aligned} \mathrm{Z} &= -b\left(\omega_1^2+\omega_2^2+\omega_3^2+\omega_4^2\right) \\ \mathrm{L} &= b\left(\omega_4^2-\omega_2^2\right) \\ \mathrm{M} &= b\left(\omega_1^2-\omega_3^2\right) \\ \mathrm{N} &= d\left(\omega_2^2+\omega_4^2-\omega_1^2-\omega_3^2\right) \end{aligned} \rightarrow \begin{aligned} \omega_1 &= \sqrt{b(-\frac{\mathrm{Z}}{4}-\frac{\mathrm{M}}{2})-d\frac{\mathrm{N}}{4}} \\ \omega_2 &= \sqrt{b(-\frac{\mathrm{Z}}{4}-\frac{\mathrm{L}}{2})+d\frac{\mathrm{N}}{4}} \\ \omega_3 &= \sqrt{b(-\frac{\mathrm{Z}}{4}+\frac{\mathrm{M}}{2})-d\frac{\mathrm{N}}{4}} \\ \omega_4 &= \sqrt{b(-\frac{\mathrm{Z}}{4}+\frac{\mathrm{L}}{2})+d\frac{\mathrm{N}}{4}} \end{aligned} \tag{1}$$

The motor values are determined by inverting these equations. Pitch (M) was reversed so a minus was used in the final formulas.

Now we can command the motors in terms of moments and forces. Note that b and d have been redefined to $b = 1/b_{old}$ and $d = 1/d_{old}$. This is due to our fixed point arithmetic having 22 integers bits and 10 fractional bits and thus multiplying will be more accurate then dividing.

3.7 Calibration

As the sensor values of the drones have a slight offset, the calibration mode is used to determine at what sensor values the drone is completely flat on the ground. These sensor values are captured 20 times and are averaged over a period of a 100 ms, These values are stored and the drone switches from calibration mode to safe mode. In either of the control modes (yaw-controlled, full-control and raw control), the stored calibration values are subtracted from the read sensor values during these control modes.

3.8 Yaw-controlled mode

This mode employs a P-controller for keeping the yaw rate $\dot{\psi}$ as close as possible to the reference yaw rate $\dot{\psi}_r$, which is set by the joystick. Important is to highlight that the yaw input sets the yaw rate and not the yaw angle, which is the case for the other two inputs, pitch and roll. The rest of the controls, pitch and roll, operate as described in Section 3.6. The P-controller operating on yaw mode is described by Equation 2. Here k_p^{ψ} is the P parameter which needed to be tuned to successfully fly the drone. In our case this was 42.

$$N = k_p^{\psi}(\dot{\psi}_r - \dot{\psi}) \tag{2}$$

3.9 Full-control mode

In full-control mode all of the Euler angles ϕ, θ, ψ are controlled by a controller. For ϕ and θ a PD-controller is used and for ψ the P-controller of Section 3.8 is reused.

$$L = k_p^{\phi}(\phi_r - \phi) + k_d^{\phi}\frac{d(\phi_r - \phi)}{dt} \tag{3}$$

$$M = k_p^{\theta}(\theta_r - \theta) + k_d^{\theta}\frac{d(\theta_r - \theta)}{dt} \tag{4}$$

$$N = k_p^{\psi}(\dot{\psi}_r - \dot{\psi}) \tag{5}$$

These PD-controllers work by adjusting the control signal on the proportion of the error and its derivative. This means that if the error is large, the proportional controller will try to steer back to the reference and when the error changes rapidly, the derivative controller will try to counteract this change to eliminate oscillations.

The different P and D gains have been found empirically. For k_p^{θ} and k_p^{ψ} we found that 40 works the best. For k_d^{θ} and k_d^{ψ} we have found 10 to work the best. Normally a D gain larger than the P gain would perform the best, but the D gain is amplified with a factor of 10 to enable larger variations in D.

3.10 Data log

To satisfy the debugging and profiling requirements of the drone function, we have implemented the function of data login, which stores the relevant real-time information that needs to be recorded in Flash. After functional mode, we can enter log mode, which disables all functions except transmission, to transmit the log information to the PC and then store it at drone_data.txt file. Hence users and developers can use these log messages for debugging or profiling. To deposit the relevant information into Flash, we need a protocol to serialize the information commands. Unlike the communication protocol, the log protocol serializes a buffer of a fixed length for subsequent reading from Flash. In addition to this, a simpler checksum is applied to validate the information, which is computed by taking the modulus 256 from the total sum of each byte in the message. The reason for this is that since it is a fixed length read/write, it doesn't require a too complex verification process as well as reducing the workload of the MCU. To prevent overflow and to determine the location of read/write messages, we set a pointer to drone, which will be updated after each read/write and will be reset to zero when the address pointed to by the pointer is larger than the maximum location of the flash.

3.11 Raw control mode

The control method of raw mode is similar to the full control mode. They all use PD controllers to adjust the motion value according to the errors and reference value. The difference is that the full control mode uses the DMP sensor value as the measured value, and the raw mode collects the sensor value from the gyro and accelerator. The structure of the Raw control mode is as Figure 6.

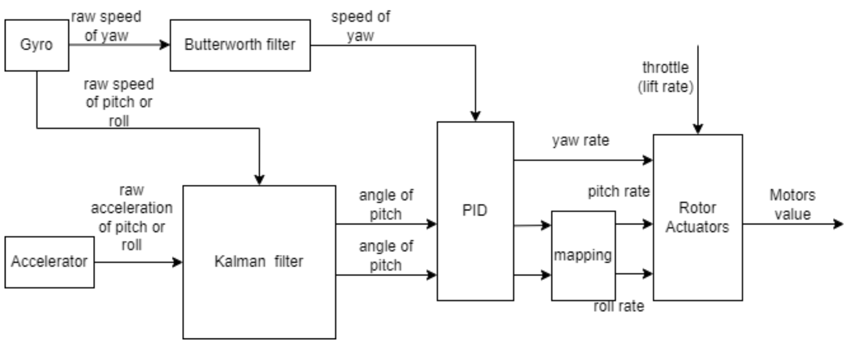


Figure 6: Raw control logic diagram

3.11.1 Butterworth filter

In this mode, we only implement the first-order Butterworth filter to eliminate the noise of the yaw signal from the gyro.

Symbol	Meaning	Expression
y[n]	current output	(N-1)/N y[n-1] + 1/2N x[n] + 1/2N x[n-1]
y[n-1]	previous output	(N-1)/N y[n-2] + 1/2N x[n-1] + 1/2N x[n-2]
x[n]	current input	-
x[n-1]	previous input	-
b_0	coefficient 1	(N-1)/N
a_0	coefficient 2	1/2N
a_1	coefficient 3	1/2N

Table 1: butterworth in ESL program

- Theory of Butterworth filter
  1. Transfer function of the first order Butterworth filter(s domain), w_c is the cut off frequency

$$H(s) = \frac{Y(s)}{X(s)} = \frac{\omega_c}{s + \omega_c}[2]$$

  2. Applying the Bilinear Transform to H(s), T_s is the sampling period

$$s = \frac{2}{T_s}\frac{1 - z^{-1}}{1 + z^{-1}}[3]$$

  3. Substitute and get the Transfer function on the Z domain

$$H(z) = \frac{Y(z)}{X(z)} = \frac{\omega_c + \omega_c z^{-1}}{(\omega_c + \frac{2}{T}) + (\omega_c - \frac{2}{T})z^{-1}}$$

  4. Z transform and then get discrete equation[3]

- Implement Butterworth filter on droneside
  To simplify the calculation process, we finally picked up the way to get the coefficients of the Butterworth filter from the lecture. N = $\frac{f_c}{f_s}$, $f_c$is the cut off frequency, f_s is the sampling frequency.

$$y[n] = (N-1)/Ny[n-1] + 1/2Nx[n] + 1/2Nx[n-1][4]$$
$$\Rightarrow y[n] = b_0 y[n-1] + a_1 x[n] + a_0 x[n-1]$$

  Based on the equation above, the program stores the previous input and output values, which can also be updated when this Butterworth filter is used.

- The result of Butterworth filter for yaw speed
  To verify the performance of our Butterworth filter, we log both the raw yaw speed and the yaw speed filtered by our Butterworth filter in Flash, then stream the log to pc side as a txt file. We use Python to draw the plot as the Figure 7. From the plot, the filtered data is significantly flatter than the raw data.

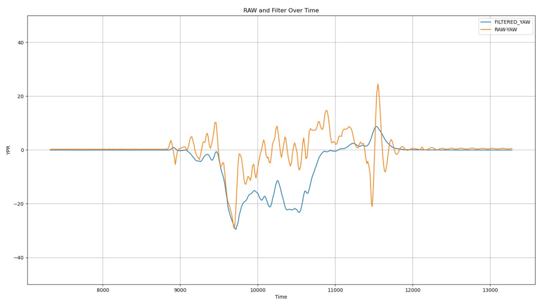


Figure 7: Effect of Butterworth filter for yaw signal(cut-off frequency:10 HZ)

3.11.2 Kalman filter

The core functionality is encapsulated within the filtering function. This function takes accelerometer and gyroscope measurements as input and updates the internal state of the Kalman filter to estimate the pitch and roll angles of the drone.
The filter combines information from both the accelerometer and gyroscope measurements to improve the accuracy of the orientation estimation. It integrates the gyroscopic measurements to track the angular rates and adjusts them based on the accelerometer readings to mitigate drift and noise.
The implementation includes simplified calculations. After performing calculations based on the theory of the Pythagorean theorem, a more accurate true value is obtained and the errors introduced by the calculation of angles exceeded the errors from directly using the values of ay and az for estimation. Therefore, further simplifications were made in the calculation process to enhance the accuracy of the actual filter.
While maintaining simplicity, the filter includes mechanisms to handle errors and noise in sensor measurements, enhancing the robustness of the drone's orientation estimation.

3.12 Height control mode

Based on the full control mode, we implement the height control mode by adding the height control function. For the height control, we need to control the lift of the drone according to the real-time height. Thus, we made a simple altimeter using a barometer, and it requires an ideal position for calibration to set the zero point of the height. After that, the drone will calculate the current altitude according to the difference in air pressure from the zero point and use the P-controller for feedback to stabilize the system. Due to timing issues, this feature was not integrated into the GUI, but it is still possible to access this mode using the keyboard key number 6.

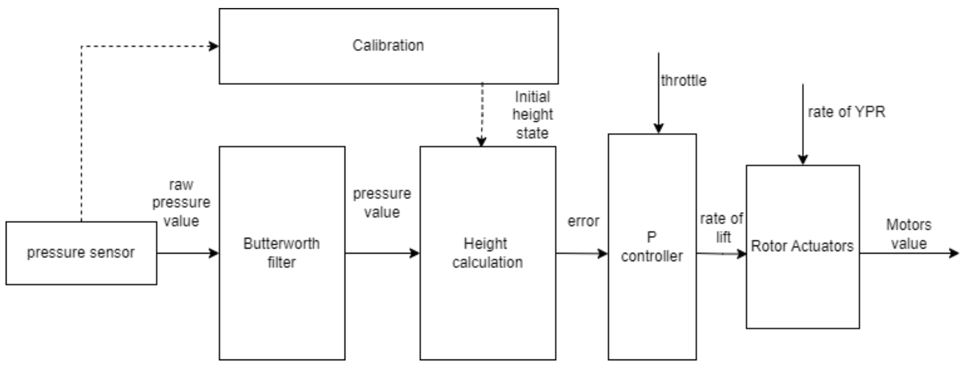


Figure 8: Height control mode logic diagram

3.13 Profiling

For measuring the performance metrics of the above features, we need to profile them. As we have already implemented data logs, by analyzing the returned data logs in this program, we can get the profiling of the corresponding functional modes.

4 Experimental results

Experimental results of the performance of our drone were measured quantitatively in metrics different parts of the design. First of all, the effectiveness of the P-controller on the drone is shown. Then, the effect of the designed Butterworth and Kalman filters on the raw sensor data is demonstrated.

4.1 Safety feature results

During the demonstration, the drone passed all safety tests. The drone can switch between different modes through the safe mode. Furthermore, the drone can enter the panic mode through esc, space, and trigger button immediately. After disconnecting the joystick or the USB cable (for communication), the drone enters the panic mode within 300 ms (which is set by the keepalive timer). The only flaw that was noticed during the demonstration is that after entering safe through panic, the joystick throttle has not returned to zero, but the drone can still enter other modes from the safe mode. However, this behaviour was inconsistent, and this phenomenon could not be replicated in other tests.

4.2 Profiling results

Table 2 shows the average running time of each loop in different modes at a frequency of 100 Hz. This includes the time to wait for the next tick at the end of each loop, proving that at a given frequency of 100 Hz, the drone can complete its function. The data in the table are averaged over 50 samples.

Mode	Period of one loop(average)
Safe	10.63ms
Panic	10.02ms
Manual	10.00ms
Calibration	9.60ms
Yaw control	9.75ms
Full control	13.44ms
Raw	13.62ms
Height control	13.81ms

Table 2: Profiling results for modes

4.3 Calibration results

Enter calibration mode from safe mode. You can see the mode switching from the GUI, and enter safe mode after completing calibration. Further operations can then be carried out.

4.4 Manual mode results

In manual mode, the drone shows excellent controllability. The drone provides correct and immediate feedback to control and external influences. During the attempt to take off, the drone took off smoothly and behaved correctly.

4.5 Yaw mode results

In yaw mode, the drone rotates based on the given value of the joystick and gives correct feedback to the influence of external forces. When the parameters of P and D control are not set, the drone will give feedback according to the manual mode.

4.6 Raw mode results

After entering raw mode, the drone cannot work perfectly like manual mode. The drone generally shows an unstable working state, but the direction of the feedback is generally correct.

4.7 Height mode results

When testing height mode, the drone cannot control itself at a fixed height. But generally it will give correct feedback to the height within a range. When the detected height exceeds the preset value, the motor speed will decrease. The motor speed will increase below the predetermined value. But like raw mode, the drone's height is unstable.

5 Conclusion

5.1 Design evaluation

The size of the compiled Rust programs are 46.8 kB for the drone, and 21.4 MB for the runner. As the program memory of the nRF51822 SoC is 256 kB, the Rust program for the drone is well within the boundaries. The runner code is executed on the host PC, which has significantly higher specifications than the embedded SoC, and the size of the runner code can therefore be neglected.

Block	Latency
Command receiver	11.32ms
Keyboard	0.0160ms
Joystick	0.0046ms
GUI	0.0017ms

Table 3: Profiling results for runner blocks

Block	Latency
Command receiver	134.08μs
Sensor reader	5811.4μs
Datalog	3399.8μs

Table 4: Profiling results for drone blocks

The latency of the code blocks are shown in Tables 3 and 4. For the drone, the execution time of the code is well within the tick frequency of 200 Hz for the drone. While this frequency exceeds the operation period of the different modes listed in Table 2, there were no discrepancies in the functionality of the drone during the demonstration.

Therefore, with the demonstration of all functionalities until the full control mode, it has been shown that the embedded systems lab has been accomplished with sufficient results.

5.2 Team results

Working as a team had many initial challenges. First of all, communication between members was difficult due to a language barrier. This deficiency in communication affected the productivity negatively, as team members did their tasks differently than instructed. By allocating someone to integrate the system, and by actively checking each others work, this issue was mitigated as much as possible. Additionally, due to the different background of the team members, it was slightly more difficult to divide tasks. One of the team members has a background in robotics, but not in Rust programming. This team member was therefore allocated to design the PD controllers. However, this caused some difficulties in the division of the tasks within the early stages of the project.

5.3 Individual performance

The performance of each individual has been sufficient. Each person completed their task and successfully produced functional designs.

Li Ou Hu : System integration; Safety requirement

Huixuan Wu : Protocol; PC side Multi-threaded; GUI; Butterworth filter; Height control

Chenruiyuan Yu : Finite state machine; GUI prototype; Yaw control; Kalman filter

Dielof van Loon : Joystick implementation; Manual mode; Pitch & Roll mode

In addition to these clearly assigned tasks, there are many optimization and debugging attempts that everyone has a lot of input into.

5.4 Learning experience

In the learning process, combining theory with practice is the most important feeling. In the implementation process of communication, finite state machine, control calculation, PID control, filter and GUI, the theoretically designed solution was greatly simplified in code implementation to provide acceptable control effects. Plans to optimize or modify prototypes based on actual test results greatly speed up the development process.

In cooperation, real-time communication and understanding of teammates' work progress are of great significance to the advancement of the project. When problems are encountered on important project milestones, seeking outside help immediately is of great significance to ensuring project progress.

References

[1] "Electrical model quad rotor uav,"

[2] M. Kamenetsky, "Filtered audio demo,"

[3] A. A. Luis F. Chaparro, *Signals and Systems Using MATLAB*. Elsevier, 2019.

[4] G. Lan, "Introduction to digital filtering,"