Group 5 Embedded Lab Report (CESE 4030)

Tongyun Yang (Tony)*(5651794, T.Yang-9@student.tudelft.nl)
Philippe van Mastrigt (4893298, p.g.vanmastrigt@student.tudelft.nl)
Giovanni Lin (4885090, g.lin@student.tudelft.nl)
Shraddha Bollapragada (5374081, l.s.s.bollapragada@student.tudelft.nl)

April 14, 2023

Abstract

This report presents the design, development, and evaluation of a quadcopter control system as part of the Embedded Systems Laboratory course at Delft University of Technology. The primary objective is to design, build, and program an embedded control unit for a tethered electrical model quadcopter, utilizing a custom PCB, sensor module, and motor controllers for stable flight. The software system is implemented using the Rust programming language, and focused on integrating various components such as physics, signal processing, sensors, and actuators. The quadcopter system is divided into three software systems: flight board controller software, host device software, and a passive communication protocol. The performance of the quadcopter is analyzed using data obtained during operation, and the project's design, team dynamics, and potential future improvements are discussed. The findings contribute to the understanding of quadcopter control systems and their practical applications in the field of embedded systems.

1 Introduction

The development and application of unmanned aerial vehicles (UAVs), especially quadcopters, for various purposes range from scientific research, surveillance, disaster relief, and recreational use. Quadcopters have proven to be particularly versatile due to their ability to take off and land vertically, hover in place, and maneuver easily in confined spaces. These capabilities make them ideal for tasks that might be otherwise dangerous or difficult for humans to perform. This report aims to describe the elements, processes, and outcomes of such quadcopter system, part of the Embedded Systems Laboratory course at Delft University of Technology [1]. The course provides students with a comprehensive understanding of the fundamental principles behind the design, construction, and operation of quadcopters. In teams of four, students acquire practical skills in electronics, embedded programming, and control systems.

The primary objective is to design, build, and program an embedded control unit for a tethered electrical model quadcopter. The control algorithm must be implemented on a custom PCB containing an RF SoC, interfacing a sensor module and motor controllers for stabilization during hovering and flight. The embedded software for the flight control board must be written in the Rust programming language. The challenge lies in integrating various components, including physics, signal processing, sensors, and actuators, to develop a robust and reliable control system for a stable and maneuverable quadcopter.

This report comprises six sections. Section 2 describes the quadcopter's system, block diagrams, state machines, software, and control loop parameters. Section 3 discusses design execution, challenges, team member contributions, and Rust code details. In Section 4, the quadcopter's performance with supporting data is presented. Section 5 evaluates the project's design, team dynamics, and future improvements. Appendices include component interfaces and code samples. Lastly, the figures and code sections are small to let this report stay within ten pages. This report is therefore not suitable for printing.

2 Architecture

The section covers the architecture of the system. It presents the functional block diagram of the overall system and gives an overview of all software components, as well as lay out some specifics of the drone and its code. Section 3 will explain each software component in higher detail.

2.1 Overview of the quadcopter system

The quadcopter system can be divided into three separate software systems: flight board controller

*Tongyun is also referred to as Tony in this report.

software, the host device software, and the passive communication protocol. The flight board controller is responsible for the drone behavior, the host device responsible for requests to change the behavior and communicating with the user, and the communication protocol, although not active but used by the former two, defines how communication is possible. In the report these will be referred to as the device or the drone, the host or the PC, and the protocol. The control loop on the drone runs at 150Hz and the communication is done by polling a buffer. The commands are set to be sent at 100Hz from the host. The project code consists out of 7045 lines of Rust code, obtained through running the command "git ls-files | grep '\.rs' | xargs wc -l".

2.2 Software components

On the PC, there are five software components, these are the Joystick Handler, the Keyboard Handler, the User Input Handler, the UART Handler and the GUI. The joystick handler is thread which scans for inputs from the joystick, these inputs then get turned into commands and forwarded to the user input thread. The same goes for the keyboard handler, scan keyboard inputs and forward them. The user input handler turns these commands into a message for the communication protocol, which is then sent to the UART handler. The UART handler takes this message, turns it into bytes and then sends it to the drone over the serial USB connection. The GUI is there to display what inputs are done from the keyboard and joystick, as well as show what data comes in from the drone.

On the drone, there are eight software components, these are the UART Handler, the Command Buffer, the Command Handler, the State Machine, the General Controller, the Motor Controller, the Message Formatter and the Data Logger. The UART Handler processes incoming data from the UART and puts it into the Command Buffer, it also formats sends messages back to the PC. The Command Buffer stores data bytes until enough bytes have been collected to make a whole data packet. This data packet is then turned into a command by the Command Handler. The command created contains the next state of the drone and the control values from the user input. The next state is passed onto the State Machine to transition to the next mode. Based on this mode, a function is called to execute actions related to this mode. If the mode is an operational mode, then the General Controller and Motor Controller are used to operate the motors of the drone. The General Controller is used for control modes, it calculates the motor compensation to achieve stable flight. The Motor Controller is used to set the motor values of the drone, these values are calculated from the incoming commands from the user combined with the motor compensation, if applicable. After the drone has processed the command, the Message Formatter makes a message that is sent back to the drone. This message contains the mode and all the sensor data. The UART Handler gets this message, turns it into bytes and sends it to the PC over the UART. The message is also logged by the Data Logger, which takes the message and stores it as bytes on the flash memory. A whole overview of the whole architecture can be found in Appendix A.

3 Implementation

This section explains the implementation of each software components, with at the end a list containing the division of work.

3.1 Communication Protocol Design

The protocol is the critical part of the whole system, it transmits commands and data. The system consists of two parts, PC and drone, therefore two protocols are designed. The reason being that commands sent to the drone are very different from the data received from the drone, therefore independent protocols would keep the system simple and clear. The two interfaces are called HostProtocol and DeviceProtocol. Such as their name, they represent where the data is sent from. HostProtocol is sent from host, also known as the PC, and the DeviceProtocol is sent from the device, also known as the drone. The next decision made is to have fixed size packets for both protocols. It is concerned that variable packets has its benefit, however, since plans have been made to write serialization functions on our own, fixed packets would be easier.

This project uses fixed packet size for both protocols, where the HostProtocol is 12 bytes large and the DeviceProtocol is 52 bytes large. The DeviceProtocol contains all sensor data, yaw pitch roll values, a start flag and an end flag, a crc and an acknowledge byte. For the HostProtocol, it is best to keep the packet as small as possible to keep a high data rate, thus there is only a single byte for each of the variables that need to be controlled and adjusted. The design choice is made to let the joystick values range from -1 to 0 then to 1 on each axis. This makes it so the joystick

still delivers proper values, but these values are partitioned combined with clamping to the nearest border value. Take, for example, a continuously variable transmissions (CVT) engine and a dual-clutch transmissions (DVT) engine. Both have values that vary from 0 to 7000 RPM, but the CVT engine is more like a continuous variation, you could make it stop at any desired RPM, while DVT only has 8 ranges, and each represents 0, 1000, 2000,... 7000. This project uses a similar system for the joystick values.

The last important design choice to mention for the protocol is that serialization crates are not used. The initial design started with serializing the message, but it has been adjusted due to reasons as follows. First, designing the protocol involves deciding the start flag, the end flag and other important bytes attached to the message. With the use of the serialization function, postcard and serde none of these are necessary anymore. The serialization process would add its own start flag and end flag to the message itself. Then by calling the deserializing function, things such as message corruption or incomplete packets are handled by the libraries. As an Embedded Systems course, it is important to want to learn how to apply these techniques, but also to know how these techniques are used behind all those functions and crates used. Moreover, "broken" messages have been observed at the beginning of the implementation of the protocol. What would happen is that a message of 52 bytes would be broken into two parts, which is unintended. The process to debug this is quite difficult due to having to work with raw byte data. In short, due to a purpose of learning more about the design of real protocols and debugging, the chosen method of transmitting data is in raw bytes.

3.2 User Input

This section discusses the implementation of the user input, so how control commands for the drone are handles. In the case of this project, it represents data transmitted from both the joystick and the keyboard to the drone. Three threads are created for this purpose (five threads in total on the host side, one for communication, and another one for the GUI). The thread structure is as shown in Figure 1.

The three threads are "User Input", "Keyboard" and "Joystick". The user input thread is where commands are formatted into a HostProtocol data packet. This packet is sent through a channel to the UART Handler. This implementation is realized by having one single HostProtocol. Whenever inputs come through to the user input thread, the corresponding instruction is sent to update the HostProtocol. The keyboard thread is where the input on the keyboard is being monitored and processed. When a keyboard input comes in a corresponding instruction is sent to the user input thread, through a channel as well. The joystick thread functions in a similar way, but there is a catch. Instead of sending instructions to update the HostProtocol, it directly sends the data received from the joystick, to the user input thread also using a channel. Due to the structure of the inputs, the effect of a key being pressed is permanent, but moving joystick only delivers temporary data. Therefore it makes more sense to immediately process and send this data. Additionally, instructions sent by the keyboard thread are realized as enums, while the joystick thread sends a struct that contains the temporary data coming from the joystick.

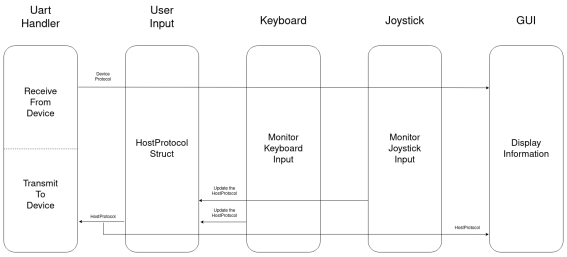

Figure 1: Thread Structure on PC

3.3 Communication

The communication consists of two parts, the host (PC) and the device (drone). The host mostly consists out the UART handler thread, the rest are some design patterns which deal with certain issues, such as broken[1] messages, command transition frequency and more. The device only consists out of a message handler with command buffering. The command buffering is done to decide when to drop one single command, when to clear the whole command buffer, and how to deal with broken commands.

As shown in Figure 1, receiving messages from the device and sending messages to the device are realized in one single thread. The initial plan was to have separate threads for these two purposes. This, however, raises an issue of sending and re-

[1] "Broken" in the case of this project does not mean "corrupted". The messages are still valid, but it means that the UART is read while it is not yet done writing a full message. Therefore, the remaining part of the message is received with latency.

ceiving at the same time. This should not be possible, because reading and writing are done via two different "paths". The solution for this is to have an UART handler thread which either sends or receives data. This is achieved by calling the function which tries to receive data from device. If there is something coming in, then go through the validation process of the message and display it, else format a message and send a packet to the device. To deal with the so called "broken" message, a message byte counter is kept track of and a set of flags is updated while receiving data from the device. A detailed example of this would be, start from the very beginning of the UART handler and try to receive data from the device. If there is data coming in and the first byte of data that comes in matches the start flag, start receiving by setting the start receiving flag to true and increment the message counter. Every time a byte gets pushed into the message buffer, the counter is incremented. When a packet is successfully received, the counter value is equal to the defined fixed packet size. If the value is larger than desired, then the packet is seen as corrupted and it will be dropped. If the value is lower than desired, try receiving bytes from the device again, and set the repeat flag to true. The program repeats the process mentioned above. The repeat flag states that "In order to receive this message, UART has already been read twice". To minimize the receive message from device time, at most two read from UART actions are be performed. After reading sufficient data from the UART (twice if necessary), the value of the message counter is checked. Only if the counter value is equal to the desired message length, then it is passed to the process of message validation. Else, it is considered a "garbage message". However, "garbage messages" can never occur, due to the low frequency of receiving messages from the drone.

The same implementation for receiving messages exists on the device. However, the device receives commands at a much higher rate than the host, therefore a different strategy is applied. A command buffer is created on the device, every 6.67 milliseconds the device tries to receive commands from the host. If there is any incoming data, it gets put into the command buffer. The bytes in the command buffer are only handled after its length exceeds the defined packet size. If the start and the end flag do not match to their appointed values, the command buffer is directly flushed. This works, since a new command is expected to be received in 10 milliseconds, and if one packet corrupts, then the next incoming packet should be processed as soon as possible. If the start and end flag of a message match then pop bytes from the command buffer until the position pf the verified end flag. Then, the packet is passed into the the validation process. The rest of the data remains in the buffer until a full packet size is achieved again.

3.4 State Machine

The implementation of a state machine is a crucial aspect of drone control, as it modulates the mode of operation. In this project, we have followed the state machine design presented in the lectures, with some minor modifications. We have defined five primary states for the drone; one of these states includes multiple sub-states.

Safety Mode: This mode prohibits any movement by the drone and disregards all movement commands from the joystick and keyboard. The drone always returns to safety mode before entering any other state, rendering it the base state.

Panic Mode: This mode facilitates the safe transition back to safety mode. After operating the drone, a command can be delivered to initiate panic mode. This command systematically reduces motor power, and upon powering down, the drone reverts to safety mode.

Manual Mode: This mode is distinct due to its operational nature; sensor data is not utilized, and the drone does not require calibration. Manual mode is treated as a separate state. Prior to transitioning from safety mode to manual mode, the drone verifies if the joystick is in a neutral position. If not, the transition to manual mode is disallowed.

Calibration Mode: Through this mode, the drone calibrates its sensors and stores the baseline sensor values. Calibration establishes a reference point for subsequent drone computations during operation mode. Upon successful calibration, a flag is set to indicate operational readiness. The drone cannot enter the operational state unless calibration is complete.

Operational State: This state comprises multiple sub-states, including yaw control mode, full control mode, raw sensor reading mode, height control mode, and wireless mode. Before entering the operational state, the drone must be calibrated, and the joystick should be in a neutral position. If these criteria are not met, the transition is denied. Each substate serves a specific purpose, such as controlling yaw or pitch and roll,

maintaining height, or processing raw sensor readings.

Data Logging State: This state is responsible for transmitting data from the flash memory of the device to the host. This state can only be accessed from the safe state.

Upon receiving a packet from the PC, the drone determines the mode and passes it to a state transition function, which validates the transition. Once the transition is approved, the on-drone state machine's state is modified. Subsequently, the transition function is executed. For instance, if the next mode is yaw control mode, the drone checks if the controller is neutral and if calibration has been completed. Upon meeting these conditions, it enters the yaw control mode, and the corresponding functions are executed. The state machine diagram in Appendix B provides an overview of the entire state machine.

3.5 Manual Mode

Manual mode is a "test mode" where the user inputs can be visualized. It also includes the testing for all safety functions.

In this mode, the user input is directly mapped to the values set to the motors. E.g. The lift command the user inputs is mapped linearly to a range from 200 to 350, the yaw command is mapped to a range from -80 to 80, etc. These values are then calculated using the equations below in order to determine the values that each motor should be set to.

```
let ae1: u16 = (lift - pitch - yaw) as u16;
let ae2: u16 = (lift - roll + yaw) as u16;
let ae3: u16 = (lift + pitch - yaw) as u16;
let ae4: u16 = (lift + roll + yaw) as u16;
set_motors([ae1, ae2, ae3, ae4]);
```

3.6 Calibration Mode

In calibration mode, an offset for sensor values from DMP and pressure sensor is calculated. They each corresponds to a control mode, i.e. yaw, pitch and roll offsets target yaw mode and full control mode, pressure offset targets the height control mode. They all aim at creating a "zero set point" for the drone, namely, after the calibration, the drone should also consider the position it has been calibrated as the set point. Note that the calibration of the pressure offset is more complicated, hence it requires more explanation, and it will be presented in detail in the section of height control mode.

This is realized with the struct "SensorOffset", which consists of 4 offset values, each corresponding to a parameter that require control and a sample counter.

When the drone enters calibration mode, it would continuously update each parameter in the struct by adding the current sensor data. The offset is calculated when the user manually transits the drone state to safety mode. It is convinced that the more samples used to calculate offset, the more accurate the offset would be. Hence, this is a design choice we made by passing the user the amount of samples they would like to use for calculating the offset.

However, this is not yet the end, once the calibration mode starts, a moving window averaging filter also starts to process the "zero set point" of the pressure data in order to minimize the effect of noise.

3.7 Yaw Mode

Yaw mode is very similar to manual mode, in fact, it is modified based on manual mode. The difference is that a "yaw compensate" value is added to the equations as shown in manual mode.

The compensate value is determined by a proportional control loop with the yaw rate from the joystick as set point, and yaw angle from the DMP as input.

3.8 Full Control Mode

Full control is built using yaw mode as a basis. Besides a yaw compensate value, it also determines a roll and pitch compensate value respectively. All compensating values are added to the equations shown in manual mode.

The compensate values are determined via different control loops. As mentioned in the section above, the yaw compensate value is determined via a proportional control loop. The roll and pitch values are determined via two cascaded proportional control loops that share the same proportional values.

3.9 Raw Mode

This sections covers the quadcopter's raw mode. This means that the quadcopter relies on external sensor fusion instead of the internal digital motion processor (DMP) of the MPU6050 board. The primary objective is to estimate the quadcopter's orientation in terms of yaw, pitch, and roll angles using a first-order low-pass Butterworth filter and a Kalman filter for sensor fusion. Raw sensor readings were obtained from the accelerometer and gyroscope through the quadrupel library via the read_raw() function which gives us the raw values in degrees and degrees/sec.

Initially, a first-order Butterworth low-pass fil-

ter was employed to filter the calibrated accelerometer and gyroscope data, removing noise and stabilizing the sensor outputs. The filter utilized a sampling frequency of 150 Hz and a cut-off frequency of 22 Hz. Subsequently, the accelerometer and gyroscope data were combined using a Kalman filter, which converted accelerometer data to angles in radians, computed roll and pitch angles, and calculated the error between integrated and accelerometer-derived angles. The yaw was calculated majorly using the gyroscope data. The fused orientation data was then corrected by subtracting the bias calculated using the Kalman filter, yielding the estimated yaw, pitch, and roll angles of the quadcopter. The C1 and C2 values used were 1.5 and 4500 respectively.

Although a Madgwick filter was considered due to its computational efficiency and suitability for real-time applications [2], a Kalman filter was ultimately implemented to better comply with the course expectations. In addition, the Madgwick filter may be more susceptible to magnetic interference and abrupt movements. The implemented Kalman filter provided a decent estimation of the quadcopter's orientation.

3.10 Height Control Mode

Height control mode is very similar to yaw mode, except that a compensate value for lift is calculated via a PID controller instead of a proportional controller, with target altitude as input. The lift command from user input was directly mapped to a value that could be set for the motors, now the mapping function returns an extra parameter which is target altitude of the drone.

As mentioned previously in the section of calibration mode, a moving window averaging filter is activated when calibration mode is activated. The window averaging filter keeps on working after user exits calibration mode. It is only deactivated when the "zero set point" is removed, which is done by panicking the drone. Hence, every time the drone panics, a new calibration should be done in order to enter another control mode. The output of this filter is used as the input of the PID controller indicating the current altitude of the drone.

3.11 Data Logging

Data logging helps in the analysis and optimization of quadcopter performance. It enables a comprehensive understanding of the drone's behavior, facilitates the fine-tuning of control systems and filters, and ensures data recovery even in the event of communication protocol failure. In this section, the implementation of data logging is discussed.

The data logging system operates by recording all relevant information from the drone at a consistent frequency (e.g., 5 Hz) during its operation. This data is stored directly on the onboard flash drive, with the writing and reading addresses continuously monitored. To maintain system reliability and simplicity, the flash memory is cleared every time the drone boots up or when the flash drive reaches full capacity.

To access the logged data, an additional mode (mode 10) has been implemented, which can be activated by pressing the 'b' key on the keyboard while the drone is in safety state. It must be noted, however, that the mode values communicated are increased by 10. This allows for both the receiver and transmitter to know when the communication switches and concerns logging information or operational information. The logs are then read and exported to a comma-separated values (CSV) file for further analysis.

The data logging implementation is built upon the existing communication protocol format, allowing seamless integration with the developed DeviceProtocol and its associated checks and methods. Moreover, the additional mode for retrieving log data ensures more reliable access to the information without disrupting the drone's operation. For exporting the information into a file, a CSV library is used; see [3].

Potential improvements for the data logging are (i) instead of clearing the flash memory every time the drone boots up or when it reaches full capacity, a more efficient data storage management system could be implemented. For instance, the system could automatically overwrite the oldest data or implement a circular buffer to ensure continuous data logging and (ii) customizable logging parameters to provide the users with greater control over the data logging process, such adjustable logging frequency and data point selection. These improvement, however, require significant protocol changes and modification of the quadruple library.

3.12 GUI

The GUI is made using Rust's tui-rs crate which is a terminal user interface library. [4]

This library was an ideal choice as it is built on top of Rust's terminion which was already being used for getting user data from the interfaces (Keyboard and Joystick).

The UI takes two inputs one is the "Host Pro-

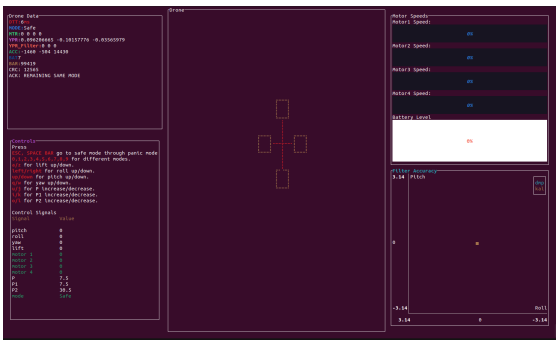

Figure 2: TUI

tocol" with all the input data from keyboard and mouse and the other is "Device Protocol" which is data received from the drone. The UI screen which can be auto-resized is split into 3 vertical sections of 30%, 40% and 30% of the full screen size at that instant.

Each vertical section is drawn using a its specific draw functions which render "Canvas" widgets like paragraph, gauge's, tables, maps and charts, etc. in the ui.rs file.

In the left most the data that is received from the drone is updated, under that the key mapping and its its current values based on keys pressed/ joystick orientation is presented. The middle portion represents the drone and based on keyboardjoystick position arrows are printed for yaw , pitch and roll directions, so it is easy to visualize in the absence of the physical drone and also if the actual drone is doing the right thing with respect to our expectations. the right side give an indication of the motor speeds in percentages with 100% being the maximum motor speed for that mode. Finally the chart represents the DMP data in blue and Kalman data in yellow when is raw mode, it can be observed that the yellow dots chase the blue dots, this is an indication that raw values are very close to DMP values.

3.13 Division of work

The division of work can be found in Table 1. The table is ordered from most work to least work. Each member is indicated by the first letter of their first name.

4 Experimental results

This section consists of the experimental results followed by an analysis of them. Gathered data as well as code for the plots in sections 4.1, 4.2 and 4.3 can be found at [5]. **Experiments done below are based on the final version used on the demonstration day with Rust code of size around 43kb, tick frequency on device at 150Hz, command frequency on host at 100Hz.**

Element	Design	Impl.	Integr.
Comm. Protocol	All	T	T
User Input	T, S	T,S	T, G
State Machine	G, P	G, P	T, G
Safety Requirements	T, G	T, G	T, G
Manual Mode	T, P	T, P	T, G
Calibration Mode	T, S	T, S	T
Yaw Control Mode	T, P	T, P	T
Full Control Mode	T	T	T
Raw Mode	S, P	S, P	S, T
Height Control Mode	T	T	T
Data Logging	P	P, G	P, G, S
GUI	S	S	S,T

Table 1: Embedded Systems Quadcopter Design Responsibilities

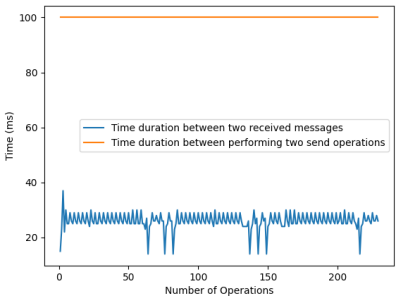

Figure 3: UART Handler Performance: Receive VS. Send

4.1 Host Communication Profile

The design choice made is that one thread on the host handles both receiving and sending messages. Therefore, it needs to be tested whether it functions like as expected.

4.1.1 Methodology Between every message received and sent the difference in time is kept track of and a plot is generated for each purpose.

4.1.2 Result Results are shown in Figure 3.

4.1.3 Analysis In the result, it can be seen that commands are sent to the device every 100 milliseconds, and that messages are received on average around every 30 milliseconds. However, this does not mean that commands are sent at a frequency of 10Hz, it means that every 100 milliseconds a set of data packets is sent, and according to data collected, on average 13 messages are sent every 100ms, resulting in a frequency higher than 100Hz.

According to other data gathered, it is also found that on average, it takes 23377.66 nanoseconds to complete the action of receiving a message, and 7117.54 nanoseconds to send a command.

4.2 Device Communication Profile

As mentioned above in Section 3.3, the decision choice was to have an algorithm to deal with re-

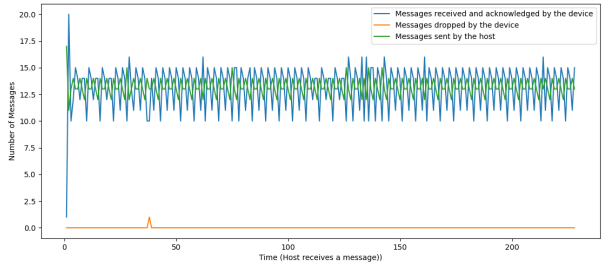

Figure 4: Comparison of messages sent by host, messages acknowledged by the device and messages dropped by device

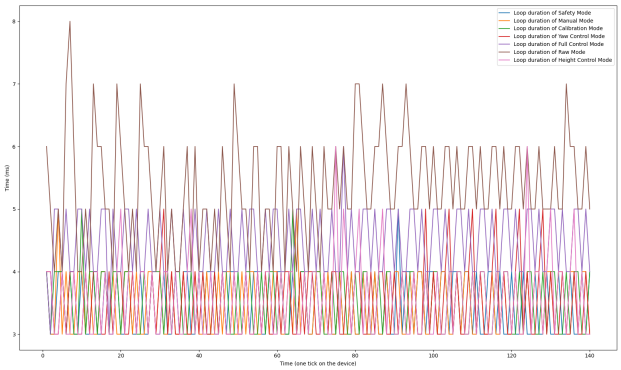

Figure 5: Duration of each mode on device

```
The average of the data in Durations/safety.txt is: 3.4571428571428573
The average of the data in Durations/manual.txt is: 3.4785714285714286
The average of the data in Durations/calibration.txt is: 3.55
The average of the data in Durations/yaw.txt is: 3.5214285714285714
The average of the data in Durations/full.txt is: 4.45
The average of the data in Durations/raw.txt is: 5.3428571428571425
The average of the data in Durations/height.txt is: 3.557142857142857
```

Figure 6: Average duration of each mode on device

ceived messages in each iteration. Hence, a visualization needs to be made of the amount of packets acknowledged and dropped during a certain time period.

4.2.1 Methodology Two counters keep track of both the numbers of times the algorithm acknowledges and drops a message. Then, the results acquired are compared with the total amount of messages expected. The experiment is repeated while the drone is operating in different modes.

4.2.2 Result Results can be found in Figure 4.

4.2.3 Analysis As shown in the results, commands are rarely dropped by the device, only 1 command is dropped among more than two thousand commands. Moreover, even though commands are grouped and sent every 100ms, it does not result in a huge latency here. Commands take time to execute, and when they are finished executing, the next group of commands are arriving. Note that the device processes the commands slightly faster than the host is sending them, hence commands would also not be queued on the device (this is shown in the following experiment).

4.3 Device Functions Duration

This section focuses on the control loop iteration time of each operation mode on the device.

4.3.1 Methodology A timer keeps track of the execution time while the drone is functioning in different modes. A plot indicating the different time durations is shown.

4.3.2 Result Results are shown in Figure 5. The calculated average values are shown in Figure 6.

4.3.3 Analysis As seen in the plot and the average execution time of each mode, raw mode has the longest control loop iteration time of around 5.34 milliseconds, followed by full control's 4.45 milliseconds, and the rest are all within 4 milliseconds for a single iteration. Combining the execution time with the amount of commands received every 100ms, no noticeable latency is introduced.

4.4 Kalman Filter

4.4.1 Methodology Gyroscopes measure angular velocity, while accelerometers measure linear acceleration. However, the MPU 6050's accelerometer can also detect the force of gravity, which can be used to determine the device's orientation relative to the earth's surface.

A Kalman filter works by taking in sensor data and using it to update its internal estimate of the system's state.

In this case, the estimated system's state using Kalman filter is compared to it using DMP[2].

4.4.2 Result Using the data logging functionality, Figure 7 shows a comparison between the pitch and roll values from the built-in DMP and the custom filter which consists of a Butterworth (1st order) and Kalman (BWK) filter.

4.4.3 Analysis In general, the value outputs of the DMP and BWK filters seem to follow the same trend. However, the BWK seems to be more sensitive i.e. steeper changes in values. In addition, a slight bias can be seen in the beginning and end of the time sampling.

5 Conclusion

In conclusion, the design and implementation of the quadcopter control system have been successful in achieving the primary objectives of the

[2]DMP sensor data fusion

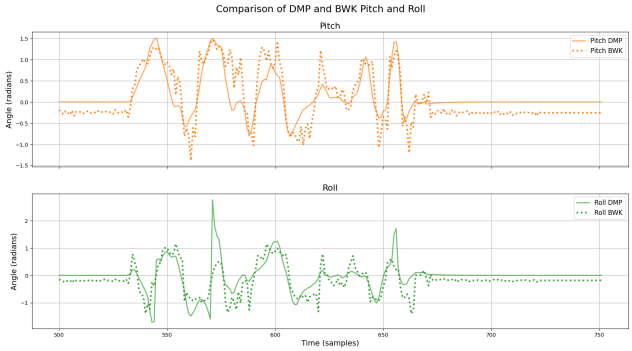

Figure 7: Comparison between the built-in DMP and the custom filter.

project. The system demonstrated effective communication between the host and device, robust state machine operation, and the ability to control the drone in various modes, including manual, yaw, full control, raw, and height control. The data logging functionality provided valuable insights into the drone's performance, while the GUI offered a user-friendly interface for monitoring and controlling the drone.

5.1 Team & individual performance

Individual performance are described by each individual respectively, followed by a concluded Team result.

Tony: In short, I've learnt a lot from the course, not only about embedded programming. I contributed to the team by taking over (sometimes helping) tasks when teammates are facing difficulties realizing, due to the fact that I might be slightly more enthusiastic and experienced in Rust. The hugest difficulty for me was to discover that the enthusiasm I have for the project is much more than others. This lead to a slightly unbalanced job division, and resulted in huge amount of time consumed. However, the bright side is that I am now more experienced in Rust programming, and I am ready to apply the skills to practice, e.g. my master thesis and further studies.

Philippe: I contributed to the team by conducting analyses, programming, organizing schedules, and facilitating team meetings and decision-making. Despite initial challenges in establishing team spirit and, later, facing obstacles in agreements on work division and communication, I focused on finding solutions and ensuring a successful project outcome. Through this experience, I gained valuable insights into embedded software engineering, control systems, and teamwork. I actively sought feedback from team members and learned the importance of open communication in future projects. All in all, this experience has enabled me to grow professionally, test personal hypotheses of better navigating team dynamics in collaborative research settings, and ultimately gave me new practical knowledge in embedded software engineering.

Giovanni: My contribution to the team was through help with coding issues, integration and debugging, and helping to realize design concepts through pseudocode and visuals. I was the only one with a computer science background, so I wanted to let the rest of the group learn more about embedded programming. When they would run into walls, I would jump in to help them. In the beginning, the group worked quite smoothly, with occasional meetings to discuss the project. However, after a few weeks there were some difficulties in the group dynamic with task division and keeping up communication between team members. Nevertheless, in the last few weeks these issues were discussed and a settlement was found between the team. It also didn't help that my grandparents got COVID during the project, therefore my parents suddenly had to leave the country to take care of them, which gave me scheduling difficulties. Due to this I would often fall behind the group in terms of progress, but I tried my best to still do what I could do. Over all, I found the project quite interesting and a great learning experience, but I wish I could have contributed more.

Shraddha: I thought the course to be an excellent chance for personal development and learning. Although I had some past experience dealing with embedded systems, I had never used Rust on actual hardware or developed a GUI. Seeing the code in action and putting control theory into reality was extremely fulfilling and added to the enjoyment of the course. Despite team dynamics issues, I kept focused on the themes I had selected to work on and felt confident in my ability to deliver alone. I carefully selected jobs that my other coworkers were not already working on to avoid disagreements and ensure that everyone felt involved. Balancing my workload while learning new ideas and ensuring everyone was on the same page in terms of deliverables was difficult, but I was able to overcome it and finish the tasks effectively. These experiences have given me significant insights that I may apply to future projects to better foresee and prepare for such circumstances.

Team: As a team, most of the required features of the drone are realized. Even though problems

occurred during the project, no one gave up and everyone sticked together to the end. Everyone also learnt a lot on both Rust programming and collaborating as a whole.

5.2 Learning experience and future work

In general, the project provided valuable experience in embedded systems design, communication protocol development, state machine implementation, and control system optimization. The project also provided an opportunity to explore the application of various filtering techniques, such as the Kalman filter for sensor fusion and the moving window averaging filter for height control.

There are several areas for future work and improvement. These include the development of a more efficient data storage management system for data logging, customizable logging parameters for greater user control, and further optimization of control loop latency. Additionally, the implementation of more advanced filtering techniques, such as the Madgwick filter, could be explored to enhance the drone's orientation estimation capabilities. Moreover, it would be great to integrate a camera to the drone for more advanced features and control.

A Overall system architecture

See Figure 8

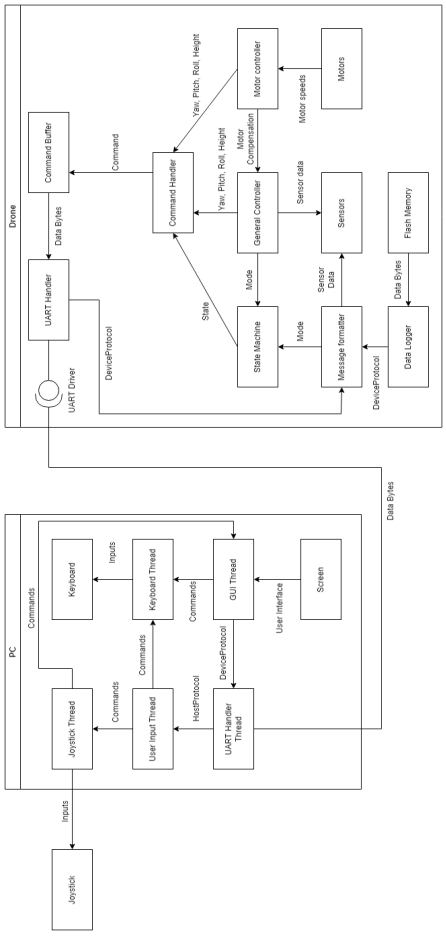

Figure 8: Overview of the system architecture.

B State Machine Diagram

See Figure 9

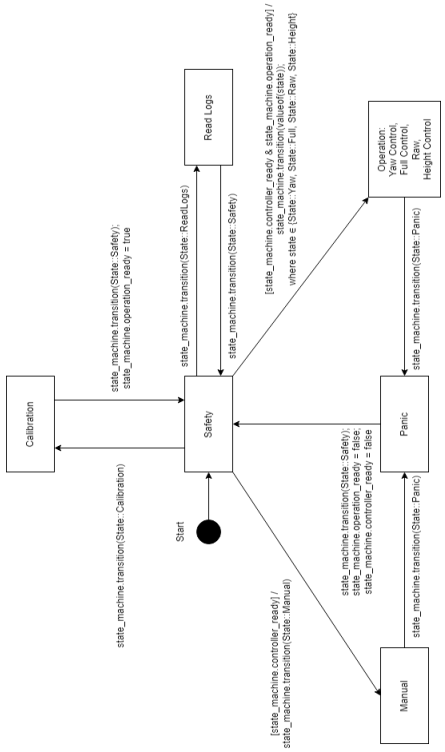

Figure 9: Overview of the state machine on the drone.

References

[1] G. Lan and K. Langendoen, "Study guide: Embedded systems laboratory," 2023. [Online]. Available: https://studiegids.tudelft.nl/a101_displayCourse.do?course_id=63221

[2] S. O. H. Madgwick, "An efficient orientation filter for inertial and inertial/magnetic sensor arrays," 2010.

[3] A. Gallant, "csv - rust," 3 2023. [Online]. Available: https://docs.rs/csv/latest/csv/

[4] "Github: Tui rust-florian dehau," (termion based rust ui). [Online]. Available: https://github.com/fdehau/tui-rs

[5] "Github: tonyyunyang/esl_pyplot," (Date last accessed 12-04-2023). [Online]. Available: https://github.com/tonyyunyang/ESL_Pyplot