Design and Implementation of a Drone-based Forest Fire Monitoring System Including an Exclusive Hardware-in-the-Loop Simulator

Hossein Jamshidi

A Thesis

in

The Department

of

Mechanical, Industrial and Aerospace Engineering

Presented in Partial Fulfillment of the Requirements

for the Degree of

Master of Applied Science (Mechanical Engineering) at

Concordia University

Montreal, Quebec, Canada

July 2021

CONCORDIA UNIVERSITY

School Of Graduate Studies

This is to certify that the thesis prepared

By: **Hossein Jamshidi**

Entitled: **Design and Implementation of a Drone-based Forest Fire Monitoring System Including an Exclusive Hardware-in-the-Loop Simulator**

and submitted in partial fulfillment of the requirements for the degree of

Master of Applied Science (Mechanical Engineering)

Complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the Final Examining Committee:

_____ Chair
Dr. Chevy Chen

_____ External Examiner
Dr. Jun Yan

_____ Examiner
Dr. Chevy Chen

_____ Supervisor
Dr. Youmin Zhang

Approved by _____
Dr. Sivakumar Narayanswamy, Graduate Program Director
Department of Mechanical, Industrial and Aerospace Engineering

_____2021 _____
Dean, Mourad Debbabi
Gina Cody of Engineering and Computer Science

Abstract

Design and Implementation of a Drone-based Forest Fire Monitoring System Including an Exclusive Hardware-in-the-Loop Simulator

Hossein Jamshidi

The purpose of this study is to design a fire detection drone system with a unique hardware-in-the-loop (HIL) simulation architecture, mainly focusing on the search and localization algorithms and simulating thermal cameras to test computer vision-based detection algorithms. The autopilot hardware has been designed exclusively for this research work. The basic flight algorithm has been implemented in the autopilot firmware. To communicate and configure the autopilot, a ground control station (GCS) is developed. The GCS exchanges data with autopilot hardware using a serial port for both telemetry and HIL data links. A game engine (Unity3D) is used for implementing the simulator's 3D graphics. To solve the rigid-body equations, the Unity3D built-in Nvidia PhysX system is utilized. The simulator exchanges data with the GCS using a UDP port. The GCS acts as a bridge between autopilot and simulator. To achieve real-time simulation performance, in most of the simulation systems and the GCS, multitasking is implemented. Furthermore, a simulated thermal camera with a raw image provider (similar to the actual hardware output) and a fire-making system in a forest-like environment has been developed to set fire to the simulated forest either at a specific location or randomly. The system consistency has been tested by performing some simulation tests and furthermore by testing the system in a real flying platform and testing the drone outdoor. Finally, the outcome of the system exhibited a good agreement with the autopilot as well as the guidance and navigation system in terms of the fire detection and positioning algorithms.

Acknowledgments

I would like to express my sincere appreciation to my supervisor Dr. Youmin Zhang. His guidance, feedback, encouragement, and patience made this research program possible. The work would not have been possible without his support and encouragement, and he was willing and enthusiastic to assist in any way he could throughout the research project.

Additionally, I would like to also thank Dr. Amin Zarareh and Mikail S. Arani, my best friends, that their help cannot be neglected, and I simply couldn't have done this without you; special thanks. I would also like to thank all my friends and colleagues from Concordia for their collaboration and discussions on the topic.

And my parents, who set me off on the road to this journey a long time ago. My greatest appreciation goes to my father, who taught me how to write my first computer program when I was just a kid, and my mother, whose support thrives me through these tough days.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivations and the goals

At the outset, the latest scientific researches show that climate change has already had apparent negative effects on the environment [1]. For example, glaciers have shrunk dramatically, ice on rivers and lakes is breaking up sooner, plant and animal habitats have shifted significantly, and trees are flowering sooner strangely. Phenomena such as loss of sea ice, accelerated sea-level rise, more droughts, and massive heatwaves are the effects that scientists had predicted in the past and are happening now. Furthermore, scientists have high confidence that global temperatures will continue to rise for decades to come, primarily due to greenhouse gases produced by human activities [2].

Secondly, the planet earth is an isolated system with a vast but limited atmosphere. Earth's atmosphere has a layered arrangement [3]. From the ground to the sky, the layers are the troposphere, stratosphere, mesosphere, thermosphere, and exosphere. A tiny layer of this atmosphere is the troposphere. Nearly all weather develops in the troposphere. Even though the atmosphere has a layered structure, they are closely coupled together [4]. An event can trigger a chain reaction that one event can lead to another one; events link climate change can be considered as a chain reaction that even a tiny change in the system can lead to a huge phenomenon. If you consider these facts, it is unavoidable that human behavior can affect this isolated system either negatively or positively. Our focus is to find a solution to affect the climate or reduce the adverse effects positively [5].

Moreover, forest fires are one of the most dramatic side effects of climate change. Not even the fire releases a vast amount of greenhouse gases into the atmosphere but also it diminish our precious forests that reduce purify our air. On top of that, it forces humans and animals to migrate their habitat and lead to food shortages (the chain reaction) [6] [7]. Finding a solution for detecting forest fire is the main motivation of this thesis.

Finally, the negative effect of climate change is indisputable; thus, we have to take some serious measures immediately. Therefore, detecting forest fires as one of the most destructive phenomena is the main focus of this research to help to take necessary measures before the fire gets outrageous.

1.2 The solution

Firstly, detecting fire in a forest needs sensors. Fire flames radiate different wavelengths, including visible light and infrared. Visible light is more susceptible to getting untraceable when there is enormous smoke, which is usually the case for forest fires. Smoke detection is not a viable approach since it may not work in a low-light situation (at night) because smoke doesn't have any natural emission that a normal camera could detect. Unlike video cameras, thermal sensors can detect fire heat day and night [8][9]; thus, they are a better choice for our goals.

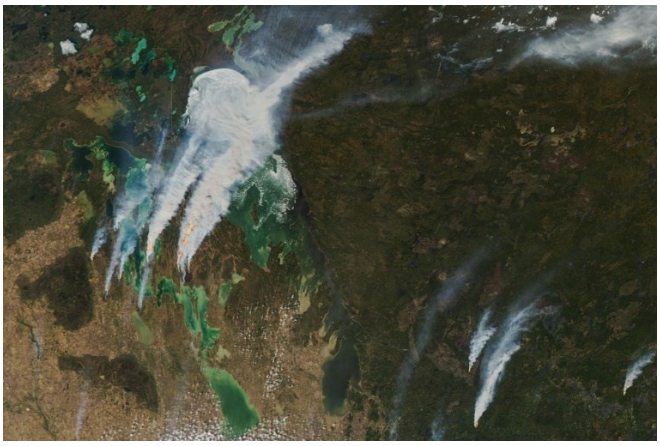


Figure 1.1: Fire and smoke at Lake Winnipeg, Manitoba, May 18, 2021 [10]

The solution should not harm nature, while the main goal is to protect it! The process of using the heat sensors in the forest must not include installing towers or modifying the wildlife

whatsoever. Given those requirements, make one of the options is very defensible, and that option is to use drones.

Drones can fly above the forest and cover a vast area while searching for any possible thermal anomaly. It is easy to deploy and, unlike maned airplanes, does not need any sophisticated infrastructure. So, using drones is an excellent approach.

1.3 Thesis objective

The main objective of this study is to design a drone capable of performing the task that is needed for fire detection, including autonomous flight and heat signature detection of the fire. Therefore, it is needed to design a flight controller, corresponding control software or the ground control station, a simulation environment to test the flight controller algorithms, and finally, test it in an outdoor environment and do the real-world experiments.

The designed system can be used for any other drones, such as fixed-wing or rotary-wing or any other robotic systems, but in order to test the system consistency, a quadcopter drone is modeled to perform the required simulation tests in the same way a quadcopter has been designed and manufactured to perform the real-world tests.

In the end, the outcome of this research is inspected be the result of both simulation (hardware-in-the-loop) and outdoor tests.

1.4 Thesis contribution

The final outcome of this research is three different products that can be used in other research too:

1. Design and manufacturing the operational autopilot hardware with all the requirements for the firmware and related software such as the ground control station. Besides, it could help other researchers to design their control boards to fit their projects.
2. A unique hardware-in-the-loop simulation, which is exclusively designed for forest fire monitoring purposes, is another research outcome. The challenges that have been

addressed during this section of research (especially for interfacing) could help other researchers that intend to do the same. The thermal camera simulation was part of this challenging section.

3. A simple algorithm is implemented to detect the fire location based on the 2D image coordination. This equation can be used for other applications.
4. Most of the academic contributions of this research, like the unique C++ objective PID controller, are closely coupled with the computer science field, and considering the software development contribution of the research is as crucial as its mechanical aspects.

Chapter 2

Autopilot

2.1 The first milestone, the autopilot

To design the autopilot, the mastermind of the flight, the final playground of our algorithm, was one of the trickiest steps of our approach. We had two options: to use an out-of-shelf autopilot and try to stitch our requirement to it or design from scratch. There some well-known open-source autopilot projects with an excellent online community that we could use; among them, Pixhawk (Figure 2.1) and Ardupilot (Figure 2.2) are the most dominant [11][12].

Figure 2.1: Pixhawk autopilot board

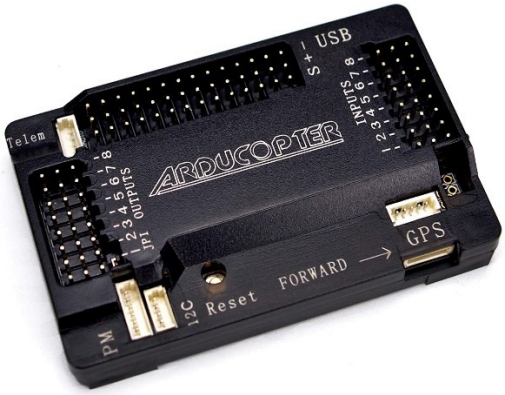


Figure 2.2: Ardupilot board

Even though they show good potentials, specially Pixhawk with a more powerful microcontroller, we decided to design our own Autopilot hardware and software from scratch. There are two main reasons behind this decision.

The first reason is to have more flexibility. When we design the hardware and the firmware ourselves, we are not limited by somebody else's design requirements, and we do not need to search among hundreds of thousands of lines of codes (that most of them are not helpful for us at all) to debug an issue. We can modify any software and hardware pieces to suit our approach that is essential at the research and development stage.

Besides, I have one visionary goal: designing the entire system to make it an industrial product and even mass-produce it that forces us to create everything from scratch and not use any particular library or anybody else's design. To use an out-of-shelf autopilot does not suit any of our goals at all.

One of the significant downsides of designing it myself is that it takes so much effort and experience. You have to read all the parts (such as sensors, microcontroller, power regulator, etc.) datasheets and application notes and apply the manufacturer's recommendation in the hardware design to avoid any elements having any interference on each other. After that, it comes to manual routing and designs the PCB tracks that need time, patience, and dedication.

If you have not had any PCB or electronic hardware design background, I do not recommend it to you and even encourage you to use one of those two options and focus on the software aspects of your approach.

The autopilot is called as AutopilotOne or APOne. The new design, APOne v1.7, is based on one of the author's older designs, APOne v1.1 (Figure 2.4 and Figure 2.3), but with a more recent and more powerful microcontroller.

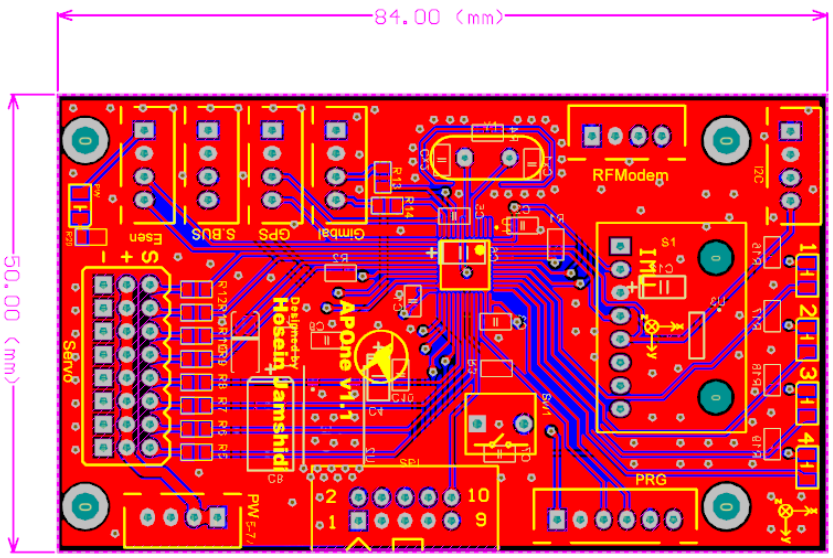


Figure 2.3: APOne v1.1 PCB layout

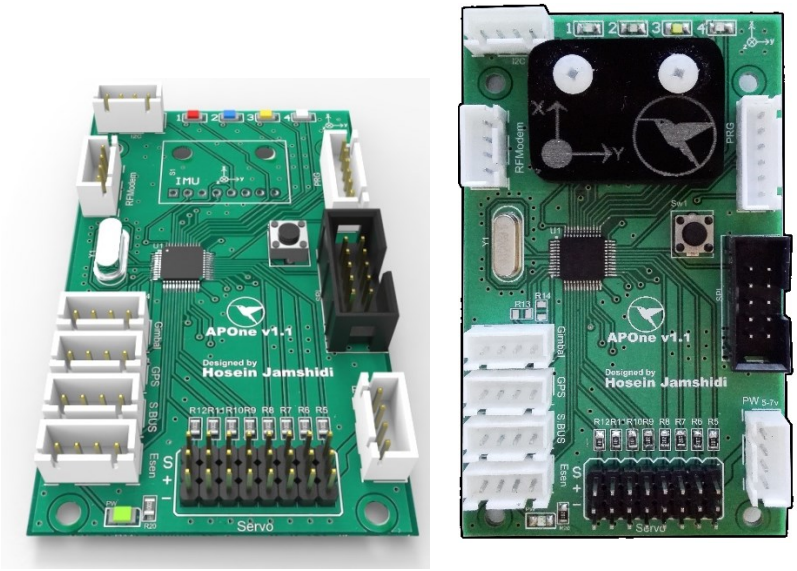


Figure 2.4: APOne v1.1 board and the PCB layout

The new autopilot has some requirements:

Based on our approach, we need to design a multipurpose autopilot that could control different platforms, including fixed-wing and rotary-wing drones. It needs to have at least 8 PWM or actuator control output. Having a three-axis accelerometer, three-axis rate gyroscope, three-axis magnetometer alongside a barometric pressure sensor. The accelerometer, gyroscope, and magnetometer are used to estimate roll, pitch, and yaw states (that we elaborate the methods);

therefore, it should be able to estimate it at least 1000 times a second; in other words, the sensor refresh rate should be greater than 1000Hz. It must have a serial port for GPS, another one for RF modem. Since designing and integrating a hardware-in-the-loop (HIL) simulation is part of our approach, we need to consider a dedicated serial port for the HIL interface alongside a powerful debug port.

2.2 Selecting microcontroller

After defining the criteria and requirements of the system, the second step of designing an autopilot is selecting the microcontroller. APOne v1.1 uses an STM32F103CBT6 microcontroller. This microcontroller has a Cortex M3 core [13]. Even though the processing power is acceptable, the new microcontrollers such as Cortex M4 and M7 have a significant superiority: Floating Point Unit (FPU). FPU handles all mathematical calculations involving floating-point variables such as float (single-precision floating-point format) and doubles (double-precision floating-point format). Most of our controller and state estimator calculations involve some double-precision floating-point variables; thus, it will significantly improve accuracy and speed if we use a microcontroller with FPU. Since Cortex M7 (STM32F7 and STM32H7 series) is a newer design, I decided to use a STM32F722RET6 as the microcontroller (Figure 2.5). The microcontroller has a 216MHz core clock, 512KB flash memory, 256KB of RAM, 8 Serial ports, 5 SPI ports, 3 I2C ports, 24 channels of 12-bit analog to digital (ADC), and numerous Timers (For PWM or DShot), which provides us a solid performance [14][15].

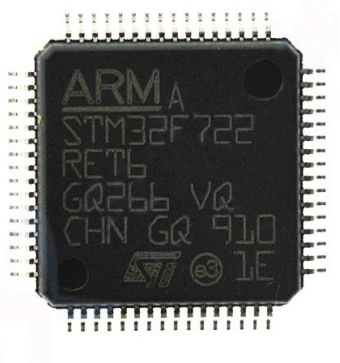


Figure 2.5: STM32F722RET6 Microcontroller

2.3 Selecting sensors

We should determine our platform attitude, heading, altitude (from sea level), and location. Thus we need to choose proper sensors for our flight controller.

2.3.1 Attitude, heading

To estimate the attitude and heading of the platform, we need to have a three-axis accelerometer, a three-axis gyroscope, and a three-axis magnetometer. The sensor that has been chosen at this stage is a TDK-Invensense MPU-9250 (Figure 2.6 and Figure 2.7). It has all the above sensors in a small package. It can calculate up to ±16g acceleration in XYZ direction, ±2000 °/sec of angular rate in the three directions, and ±4800μT of the magnetic field around the sensor in three directions [16]. We have to keep in mind that those are the absolute maximum of our system; if we go more than that, it will saturate our sensors, and our data will be inaccurate. With this sensor and a decent algorithm, we can estimate the attitude and heading more than 1000Hz. The interface could be both I2C or SPI. SPI is suitable for our design since it is faster and more reliable since it uses a separate data line for TX and RX (full-duplex mode) [17].

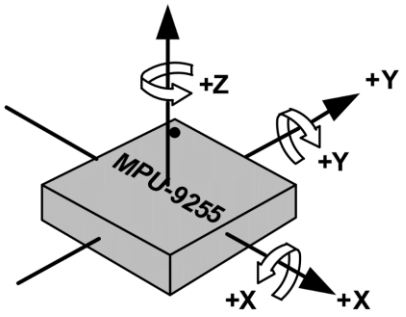


Figure 2.6: Axis of the Accelerometer and Gyroscope

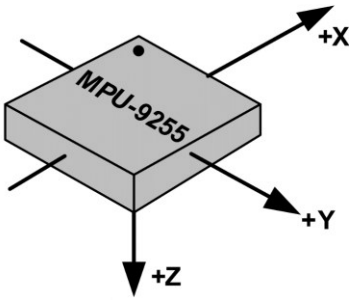


Figure 2.7: MPU-9250 Magnetometer Axis

2.3.2 Altitude

To detect absolute altitude from the mean sea level, we need barometric pressure and the temperature. After reviewing numerous available sensors in the market, I decided to choose Bosch Sensortech BMP280 (Figure 2.8). It can sense from 300hPa to 1100hPa (the equivalent of -500m to +9000m above sea level) and has an internal temperature sensor with 0.01°C resolution. The relative accuracy of the sensor is ±1 meter. The measurement rate of BMP280 is up to 157Hz [18]. The interface could be either I2C or SPI. SPI is a better choice, just like the previous sensor.

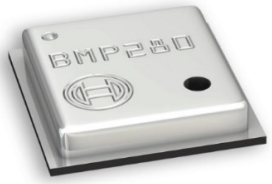


Figure 2.8: BMP280

2.3.3 GPS

The U-Blox SAM-M8Q GNSS receiver has been chosen for this platform (Figure 2.9). The module can provide concurrent reception of up to three GNSS systems, including GPS, Galileo, and GLONASS. The horizontal position accuracy of SAM-M8Q is 2.5meter and can offer up to 18Hz [19]. Because we are at a historical time that the L5 GPS signal will be available for public usage soon, I recommend a newer module that receives L5 signals for the new design. The L5 signal offers 0.3meter accuracy and more robustness and reliability. At the time of writing this thesis, the L5 signals are considered pre-operational [20].

Figure 2.9: SAM-M8Q GNSS module

2.4 Designing the PCB board

2.4.1 Schematic design

To finalize the autopilot hardware design, we need to design the printed circuit board (PCB). The first step is to design the schematics. The schematics are about the connection and relation of each element to the other components. For example, how the power system elements are connected to each other and how it will power up the microcontroller and the sensors (Figure 2.10) or allocate our microcontroller pins (Figure 2.11).

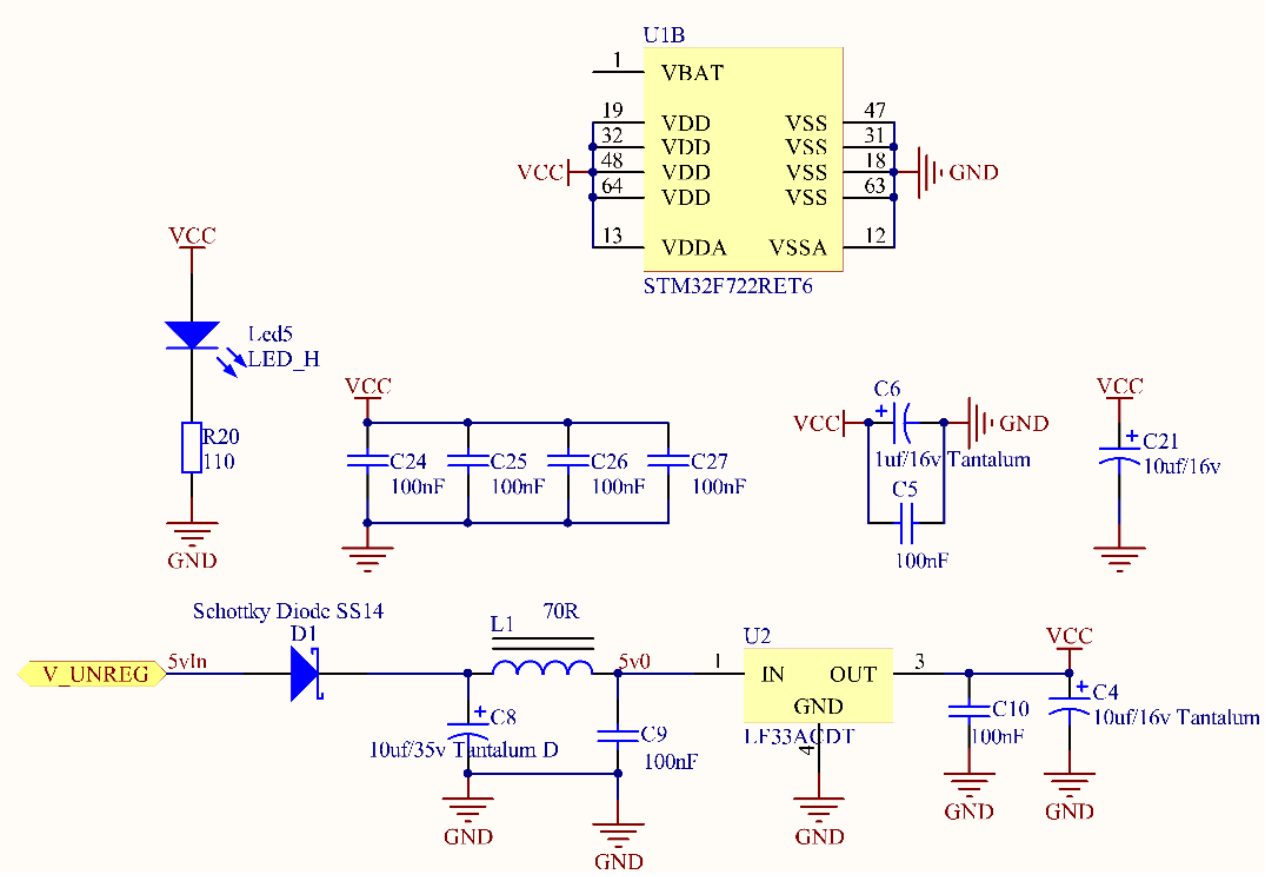


Figure 2.10: The APOne 1.7 power system

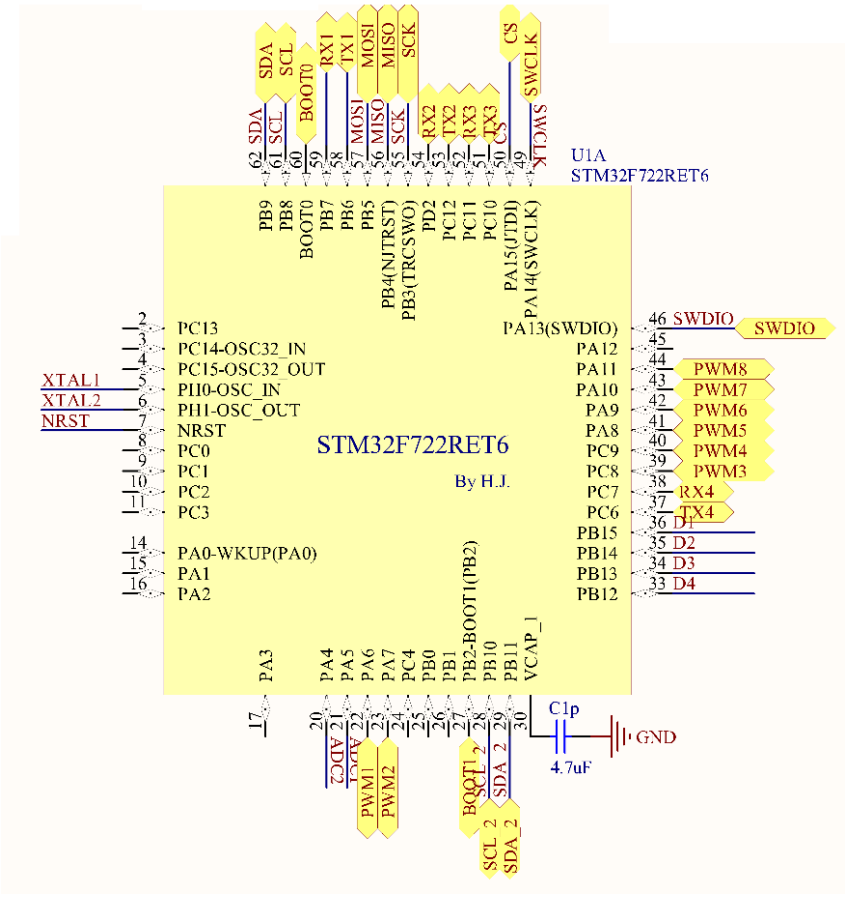


Figure 2.11: APOne v1.7 Microcontroller Pin Allocation

On top of that, we have the microcontroller, which is the most sophisticated part of our hardware. Microcontroller schematic design takes a comprehensive study of the manufacturer datasheet, reference manual, and application note [15] for that specific model. To talk about the details are out of the scope of this thesis.

2.4.2 PCB design

After designing the schematics, we need to convert them to a physical entity. We must first define our board perimeters and size and then juxtapose each element regarding their functionality and relations. After this step, the board will be something similar to Figure 2.12. To finalize the board, we must draw the physical tracks (Figure 2.13). We must consider every single track's maximum current, voltage, frequency (if it is a signal), clearance, and board impedance.

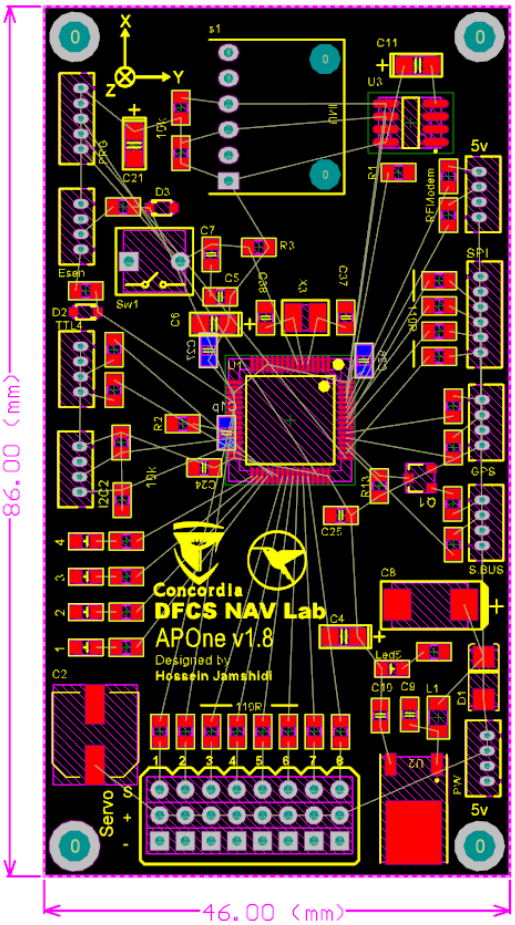


Figure 2.12: APOne v1.7 PCB before connecting the components

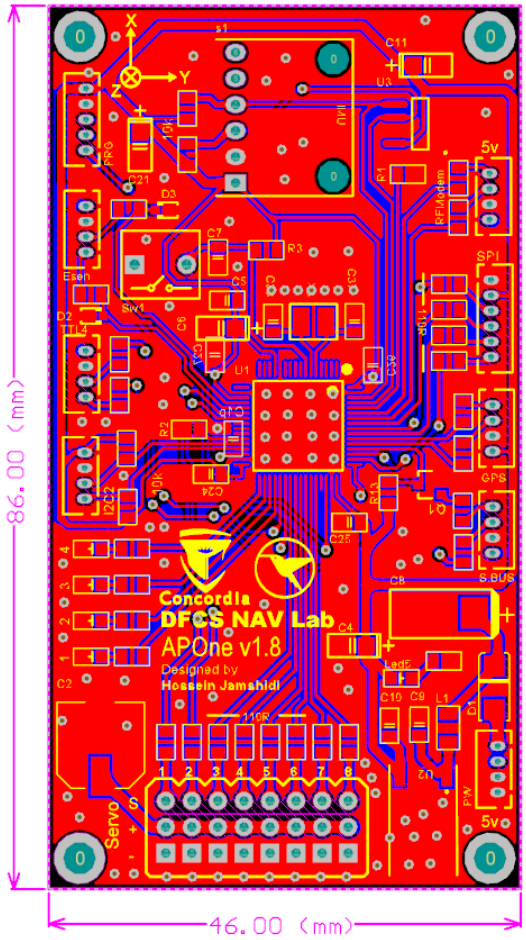


Figure 2.13: APOne v1.7 complete PCB

2.4.3 Manufacturing and assembly

After finalizing the PCB board and reviewing it carefully, we sent the file to be printed (Figure 2.14). The next step is to assemble the parts on the PCB (Figure 2.15). This step takes a lot of soldering has to be done by professionals. It is possible to use surface-mount technology (SMT) machines to assemble it, but it will be too expensive for this project. Therefore, all assemblies were done by the author of the thesis manually (Figure 2.16).

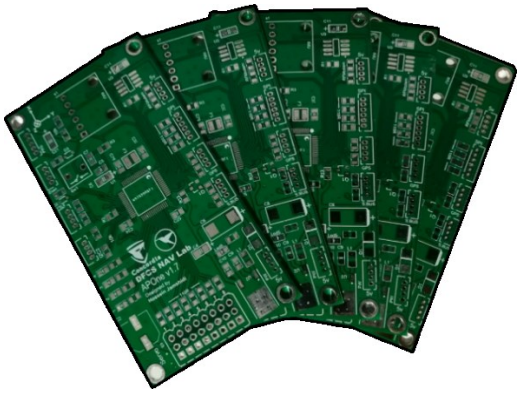


Figure 2.14: APOne v1.7 Printed board before assembly

Figure 2.15: Final board

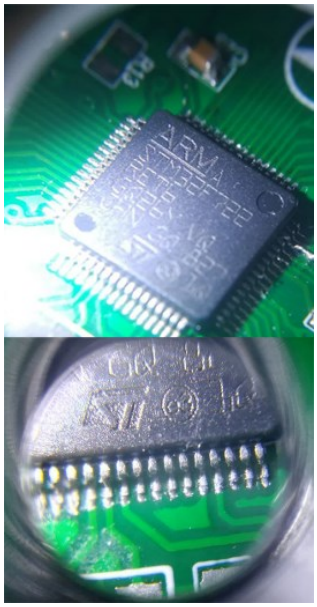


Figure 2.16: APOne v1.7 Board under a microscope

2.5 Summary

In this chapter, the procedure of designing and manufacturing the autopilot hardware is described. First, we define the criteria and then selected the microcontroller, sensors, and other parts. We design the system schematics base on each part's characteristics and behaviors. The next step was designing the PCB. After designing the PCB, we sent it to be fabricated and then assembled. This process was briefly explained, and some of the details about the board design and the electronic aspects were left out to make it easier to read. This autopilot design was based on an old design and the result of several years of research, development, and industrial experience of the author in the electronic field. We do not recommend going through this step if it is you do not have related experience.

Chapter 3

Ground Control Station

3.1 The definition of the ground control station

A ground control station or a GCS is software that communicates with the autopilot using a wireless data link. It sends and receives configurations, commands, and the system status. Our GCS is a windows application that has been written in C# using Microsoft visual studio.

The design criteria and desired functionality are as follows:

1. It must have a serial port interface to connect to the RF modem.
2. Sending the configurations such as mission waypoints, PID controller coefficients, the Output channels trim, etc.
3. It visualizes data, for example, a speed indicator or an artificial horizon.
4. It most records the sensors' data for further analysis in a proper format.

Besides that, in our implementation, the GCS also controls the HIL simulation that we will elaborate on in the HIL chapter.

3.2 Serial communication

A serial port is a digital communication interface that transfers the data bidirectionally in full-duplex mode (simultaneously send and receive) sequentially, one bit at a time. Many devices are using UART (universal asynchronous receiver-transmitter) to implement serial

communication. In contrast with most of the serial communication interfaces like SPI or I2C, UART does not have a master clock source; It just has a TX line and an RX line. As their name suggests, TX sends the data, and RX receives the other side TX data (Figure 3.1) [21].

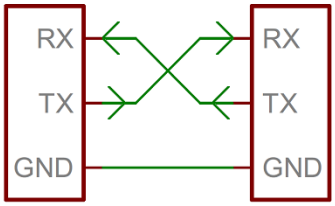


Figure 3.1:UART port connection schematic

Before starting the communication, both sides should know their baud rate to encode and decode the data properly. Baud rate is the number of bits that the UART port sends in a second. For example, 9,600bps (bit per second) or 115,200bps. A higher baud rate means higher communication speed. On the other hand, serial port communication with a higher baud rate is more susceptible to external noise and interference. In our case, we set the baud rate in the autopilot as 57,600bps(or 7,200 bytes per second); thus, we can simplify the serial port configuration in the GCS and only need to select the COM port number (Figure 3.2).

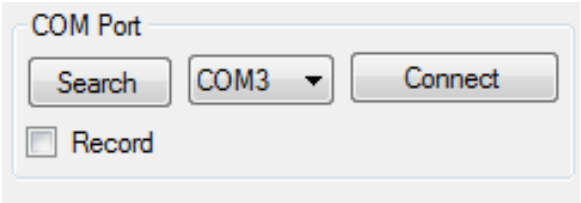


Figure 3.2: GCS Serial port connection setting

Since new computers and laptops do not have a physical serial port anymore(at least for regular laptops), a USB UART converter is used that works as a regular serial port.

Figure 3.3: A USB UART converter

3.3 Configs

An exclusive tab is implemented for the setting section to send the user configuration (Figure 3.4). It sends the user configs and can read the existing config from the autopilot to verify the sent and saved configs. The autopilot saves the config data into the EEPROM memory to keep them permanently. For example, you can read the PID controller coefficients in the middle of a flight, change them, and check the system behavior and if the autopilot gets reset, it will read the saved config from EEPROM as soon as it starts, and the operator does not need to resend the config data.

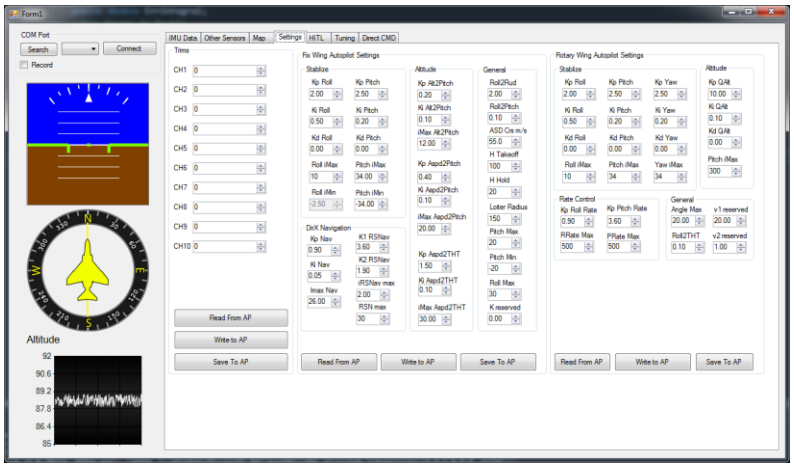


Figure 3.4: GCS Autopilot configs

3.4 Visualization

3.4.1 Indicators

Flight instruments and indicators are components design to show some flight characteristics to the users in the most intuitive way [22]. For example, altimeter, airspeed indicator, vertical speed indicator, heading indicator, turn indicator, attitude indicator, etc. (Figure 3.5 to Figure 3.10)

Figure 3.5: Altimeter

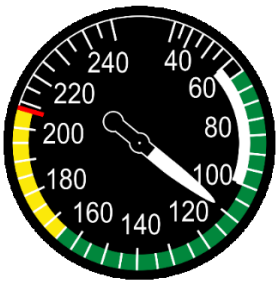


Figure 3.6: Airspeed indicator

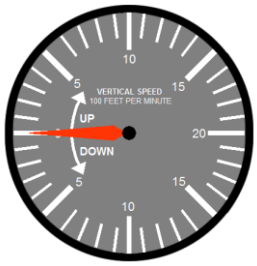


Figure 3.7: Vertical speed

Figure 3.8: Heading indicator

Figure 3.9: Turn indicator

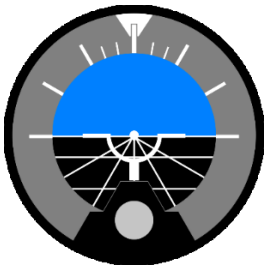


Figure 3.10: Attitude

Some of them are not necessary for our implementation, and some of them need to be changed to serve our purposes. For example, instead of showing the platform altitude as a simple gauge, for this particular purpose, it is better to show a chart of altitude that gives the user a good intuition of altitude change and its history for a short period of time (20 seconds) (Figure 3.12).

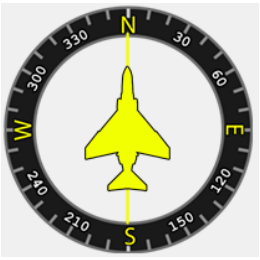


Figure 3.11: The GCS heading instrument

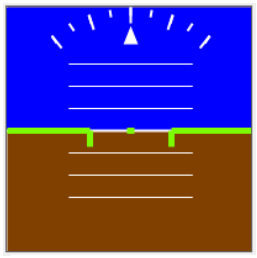


Figure 3.12: The GCS Attitude indicator

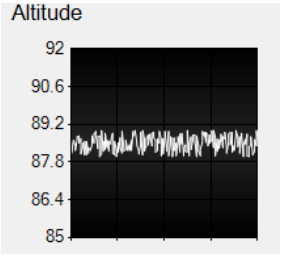


Figure 3.13: Altitude chart

The implementation of the instruments was done using C# drawing 2D that contains some advanced two-dimensional and vector graphics methods [23]. For example, you can use methods like DrawLine, DrawRectangle, DrawString, DrawArc, DrawImage, FillPolygon, etc., to directly draw your shapes into your graphical object.

The GCS attitude indicator is a sophisticated combination of trigonometric function and calculations that were developed through the course of several other projects, and to explain it is out of the scope of this thesis, but the heading instrument has a more straightforward structure, and it is helpful to describe its implementation here.

3.4.1.1 An instrument design case

To design a heading indicator, two simple images need to be created: a heading wheel and a needle. The heading wheel has to be something similar to Figure 3.14. The heading needle could be just a vertical line, or if you want to give it some style, you can add an aircraft silhouette too (Figure 3.15). Both images have to have a proper alpha channel and transparency.

Figure 3.14: Heading wheel

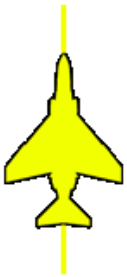


Figure 3.15: Heading needle

The method that draws the heading indicator is implemented in C# as:

```
public void drawHeading(double heading, ref PictureBox headingPicture)
{
    grHeading.Clear(Color.Transparent);
    grHeading.DrawImage(hedingWeelImage, new Point(0, 0));

    rotationAnchorX = headingPicture.Width/2;
    rotationAnchorY = headingPicture.Height/2;
    beta = Atan(rotationAnchorY / rotationAnchorX);
    d = Sqrt((rotationAnchorX * rotationAnchorX) + (rotationAnchorY * rotationAnchorY));
    deltaX = d * (Cos(heading - beta) - Cos(heading) * Cos(heading + beta) - Sin(heading) *
Sin(heading + beta));
    deltaY = d * (Sin(beta - heading) + Sin(heading) * Cos(heading + beta) - Cos(heading) *
Sin(heading + beta));
    grHeading.RotateTransform(heading);
    grHeading.DrawImage(needleImage, deltaX, deltaY, needleImage.Width, needleImage.Height);
    grHeading.RotateTransform(-heading);
```

```
    headingPicture.Image = grHeadingImage;
}
```

The first line of the method clears the graphic object and replaces it with a transparent pallet. The second line uses the DrawImage method to draw the heading wheel image (Figure 3.14). The rest of the code uses trigonometric techniques to calculate the corresponding variables for rotating and the needle image and draw it back to the center of the image. For example, the result for a 45° rotation is depicted in Figure 3.16.

Figure 3.16: The heading indication with a 45 degree heading.

3.4.2 Map and mission planner

Maps are a standard section of any GCS project. The GCS uses an interactive map to show the drone's location and setting the mission waypoints (Figure 3.17).

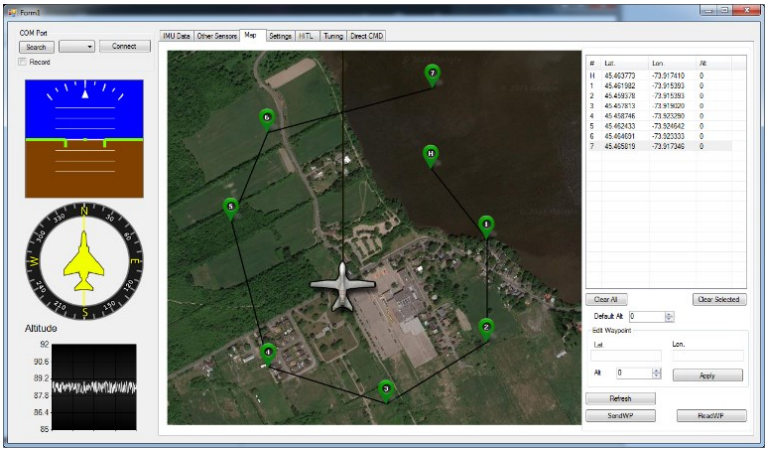


Figure 3.17: GCS Indicators and the Map

It uses some well-known map provider images like Google maps, Bing maps, MapBox, etc., to provide the map image. It also can use a still image from other satellites or aerial imaging systems. There are two critical challenges here that need to be addressed: Map calibration and data cache.

3.4.2.1 Map calibration

The GCS needs to get the vehicle coordination to the corresponding location on the map and vice versa. If the GCS Map is not accurate enough, it may show the drone location wrongly and cause some confusion, but this is not the worst side. The worst side effect of an inaccurate GCS map comes into equations when you use that inaccurate map to send waypoints to the autopilot.

The map could be calibrated using other maps and compare with the GCS map and calculate the calibration data if needed. The other method is to do it yourself and use an accurate GPS (possibly several different receivers) and determine several clear feature locations like trees or a crossroad location and calculate the calibration data. A combination of both techniques is more desirable overall.

3.4.2.2 Map data cache for offline usage

The test sites for aerial vehicles usually are outside of urban are, in some cases in remote are; therefore, the chance of having internet access or even cellular network access is meager. The GCS or all other parts of the system should be internet-independent and must be functional without internet access. The GCS map implementation should cache the map data on the hard drive. For example, it could read the map tiles while online and save them into a database as a blob and then retrieve them when the system is not online. Some map providers have restricted policies about usage and specifically about caching that all of them must be applied. For example, Google says:" Customer will not cache Google Maps Content except as expressly permitted under the Maps Service Specific Terms." [24].

3.5 Muti-threading

Multithreading is a means of doing different jobs simultaneously. Adding multithreading allows the software to run smoothly and efficiently. When the system is time-critical, and the real-time capability is in the design criteria, considering multithreading is one of the solutions [25]. Imagin this function:

```
private void serialThread()
{
    int serAval = 0;
    byte[] tmpb = new byte[1024];
    exitSerial = false;
    if (COMPort.IsOpen)
    {
        while (!exitSerial)
        {
            SAval = COMPort.BytesToRead;
            if (SAval > 0)
            {
                COMPort.Read(tmpb, 0, SAval);
                for (int i = 0; i < SAval; i++)
                {
                    // process the received data
                }
            }
        }
    }
}
```

It checks if the serial port is open then starts a loop to check if any data is available to read. If there was some available data, it will read the entire data buffer and sends it to the parser. If this method gets called in a regular single-thread system routine, it will freeze the entire software. To prevent it from happening, the method must be called on a separate thread. To run this method on a dedicated threat to the system, this part of the code is used:

```
Thread mThread;
mThread = new Thread(SerialThread);
mThread.Priority = ThreadPriority.Normal;
mThread.IsBackground = true;
mThread.Start();
```

Furthermore, it has to run separate threads for each UDP server [26] (Simulator to GCS), UART (GCS to Autopilot), and the user interface (Drawing indicators, etc.) and exchange data between each thread simultaneously (Figure 3.18). One of the standard methods to implement inter-thread communication is queue buffer. Simply it is a first-in-first-out data buffer that the provider thread writes fils the data, and the consumer threads can read the data. To talk about lock mechanism and other concurrency issues are out of the scope of this article.

To implement such complex software with a single thread solution was simply impossible or showed inferior performance, and we achieve this acceptable real-time performance by considering the parallel tasks. To test the overall performance, we designed a test to calculate the maximum data refresh rate and control loop frequency. It sends dummy data from the simulator to GCS and then to the autopilot, and it sends it all the way back to the simulator itself; this is the actual data flow of the HIL simulation. The results are shown in Figure 5.1.

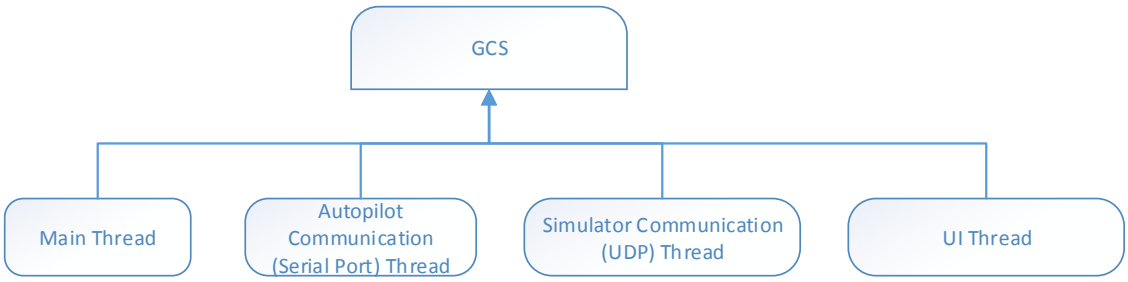


Figure 3.18: GCS threads

Table 3.1: Tasking and communication performance

Method	Delay(Average)	Refresh and control loop frequency
Single-thread	94ms	~10Hz
Multithread(shared mem)	1.6ms	~600Hz
Multithread(UDP local)	2ms	~500HZ
Multithread(UDP over ethernet)	17ms	58.8HZ

As it is shown in Table 3.1, While the single-thread performance is off the chart, the multithread implementation and shared memory as the bridge between the simulator and the GCS show superior performance. The UDP port shows a performance still in an acceptable range. We have still chosen to use a UDP port to run the simulation over a network using multiple computers for future development. Given the existing simple UDP performance, it shows promising capacity.

The rest of the GCS features, such as the data interface and the hardware-in-the-loop capability, is elaborated in their dedicated chapters.

3.6 Recording data

The GCS records the data that it receives from the autopilot. Since the data rate for different types is not equal, it makes a file for each data type. For example, the roll, pitch, and yaw are sent 50 times per second or 50Hz while it sends the GPS and altitude just 10Hz; thus, it opens a file for 50Hz data and another one for the 10 Hz data. Since it is hard to match and analyze the data in different files, it adds a timestamp to each data record. The GCS recording data format is CSV that can be read by Microsoft Excel.

3.7 Summary

In this chapter, the implementation of the ground control station is discussed. The design criteria have been described, and then described the serial port, the means of communication with the autopilot, explained baud rate, and the physical ports. The GCS sends the autopilot config that is explained how. After that, one of the sophisticated parts of the GCS, instruments, is elaborated briefly. Map and the corresponding challenges are the next topics. The benefit of multithreading was explained in the next section. The final section talks about how GCS records the data.

Chapter 4

4 Hardware in the loop

4.1 Hardware in the loop definition

Hardware-in-the-loop (HIL) simulation is a practice that is used to develop and test complex real-time embedded systems. HIL simulation offers a test base for the system by adding the difficulty of the plant under control to the test system. The complication of the plant under control is involved in the tests and developments by adding a mathematical model of all related dynamic systems. These mathematical representations are referred to as the "plant simulation" [27].

The embedded system interacts with this plant simulation to perform the tests that are hards to perform in real circumstances. For example, a flight controller test base is an aircraft; The aircraft needs to fly and perform maneuvers that are not viable nor economic by a not-yet-confirmed flight controller. HIL simulation reduces the development cost, duration, safety, and feasibility. The other benefit of using HIL simulation is to enhance the excellence of the testing by increasing the number and range of the testing. In short, In the hardware-in-the-loop test, only the plant and the feedback are simulated, and the rest of the system remains as similar as the real system.

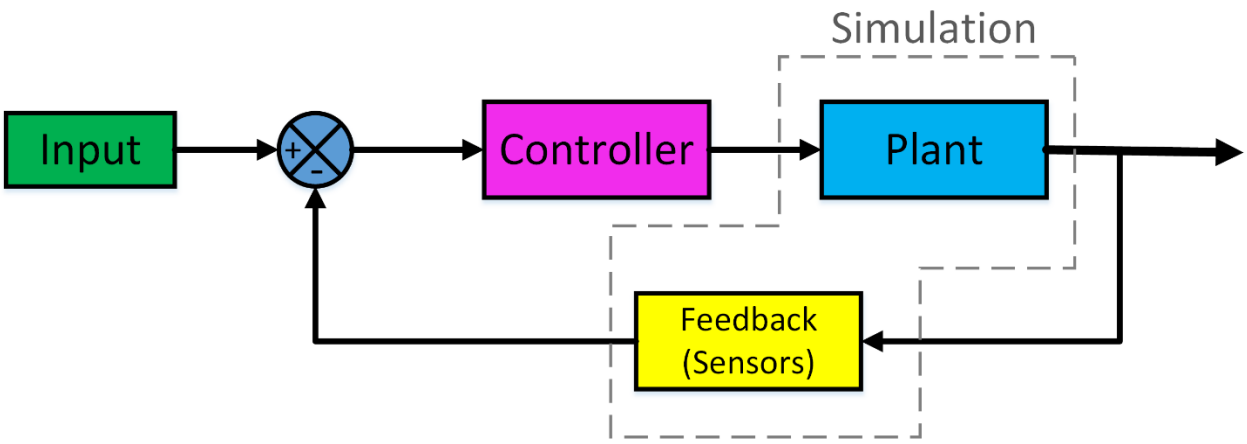


Figure 4.1 HIL simulation diagram

4.2 Plant simulation

The plant is the system and its surroundings and their reactions to each other. For example, if the system is a space heater in a room, the heater is the object. The room and every other object in the room (possibly the other heat sources) are the environments; the heater voltage is the input, and the room temperature is the output. The object and the environment are coupled together and have influences on each other. The plant that is simulated for the purpose of this thesis is a drone flying in a forest-like area with a fire in the area. For simulating the plant, a mathematical model of the object is needed. In this case (a flying object), it is called flight dynamic model. Furthermore, to make it interactive, a method is needed to visualize the environment and the plant. Sometimes it is simple; for example, in the case of the space heater, it could just show the room temperature (a simple text). But when the system is too complex, it needs some sophisticated approaches. To visualize the system behavior of a flying object, a 3D environment must be designed alongside a flight dynamic model that estimates the drone behaviors.

4.2.1 3D environment

Having an excellent real-time visual interface gives the user better insight into system behaviors. A 3D interface that shows the system behaviors in a forest-like environment in real-time is needed for this system. Such a 3D environment with this sophisticated visual aspects as forest and fire (shown in Figure 4.3) needs a complete 3D engine or a Game engine. A game engine handles all the low-level tasks, including the 3D rendering, loading the 3D models, the physics

engine, networking, memory management, threading, etc. For the purpose of this research, the Unity game engine has been chosen to implement the 3D environment because of the ease of use (especially for prototyping), more resources on the internet (3D models like trees or plants), better online community [28]. The other game engine that could have been used is The Unreal Engine that follows the same principles. The Unreal Engine takes more disk space and ram besides a long time to start; on the other hand, it has a better graphical rendering capability. Considering the pros and the cons, we decided to use The Unity engine as the workhorse behind the 3D environment. Like the designed GCS, the Unity engine uses C# as the intermediate language that helps to improve the consistency of the implementation and the codebase. Since the system is too complicated, it is better to consider a simpler environment to test the system behavior. The first stage of the 3d environment was just a simple plane with a platform for taking off and landings (Figure 4.2). A simpler environment helps to eliminate other components and parameter effects.

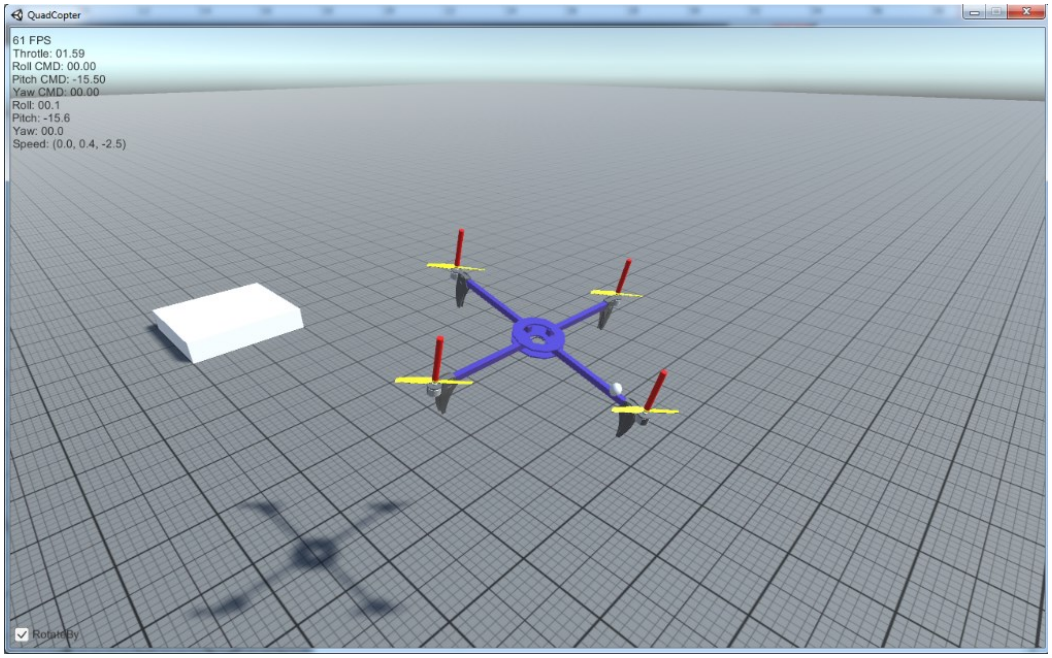


Figure 4.2: First stage of the 3D environment implementation

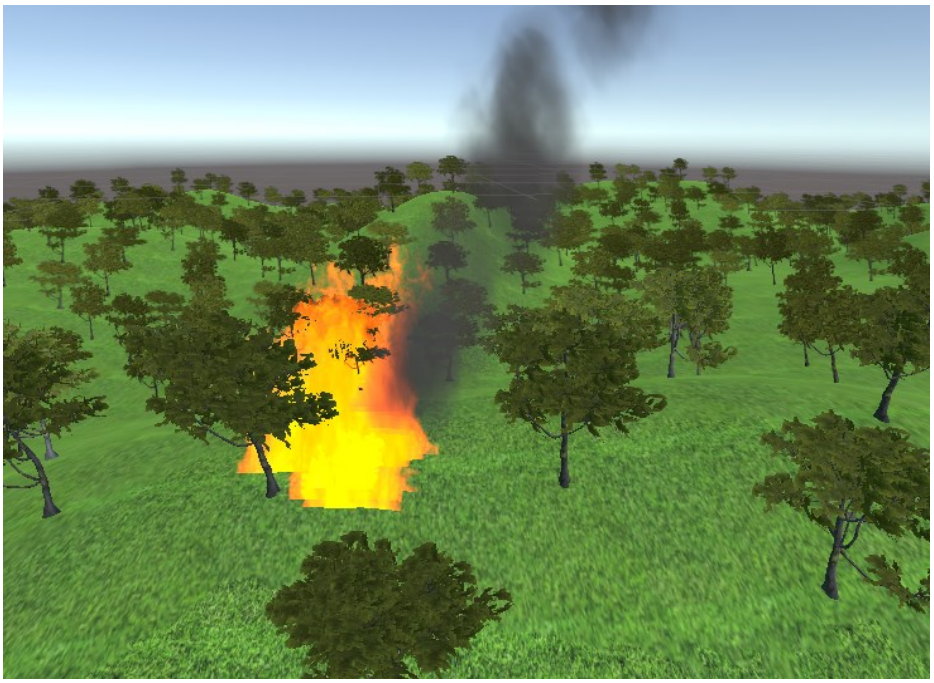


Figure 4.3: Fire and forest-like environment

4.2.2 FDM

We presume our drones as rigid-body, and to predict our system behaviors and fulfill our flight dynamic model, we need a rigid-body solver. There are many well-known implementations of flight dynamic models, especially for quadcopters [29]. In the course of this thesis, it is decided not to reinvent the rigid body solver and use a confirmed out-of-the-shelf solver in order to emphasize the efforts on the other aspects such as interfaces and real-time capability. Therefore, the generic rigid-body dynamic solver built in the Unity engine is used [30]. The Unity physics engine is an integration of the Nvidia PhysX engine. Nvidia PhysX is the leading physics engine under the hood of most of the video games in the industry and shows an astonishing performance [31]. The object definition is straightforward, the rigid-body object's physical characteristics such as weight, mass, and inertia have been defined, and then the corresponding forces and moments are applied according to the drone's nature (quadcopter, hexacopter, or even a rover) afterward the 6DOF solver estimates the object's motion and state. Figure 4.4 shows how the drone's rigid body design has been assumed. The green outlines show the actual rigid-body weight distribution. Figure 4.5 shows the corresponding force and moments regarding the actuators on the quadcopter we used as a test case. Each motor makes a lift force and a moment. In our approach, like other

popular designs, The moments have the same direction on each axis and oppose the other side to compensate for each other's momentum.

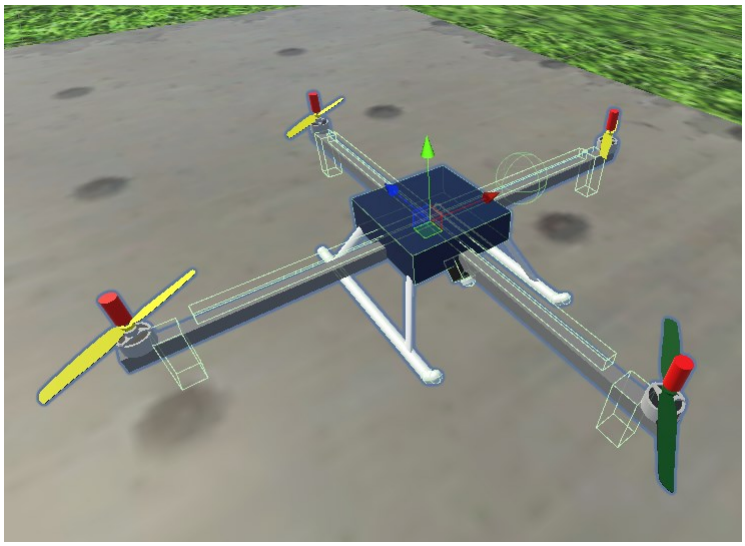


Figure 4.4: Drone rigid body design and weight distribution

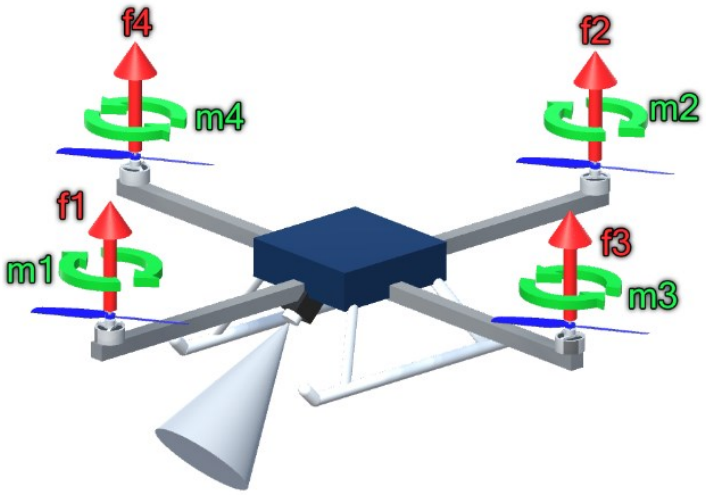


Figure 4.5: Force and moments

Besides estimating the 6DOF dynamics, our simulator can handle the object collision [32], whether it is the behavior of the ground colliders (landing gears) or the reactions between the flying objects (mid-air collisions), which is one of the key benefits of our implementation. In the other word, unlike most of the regular flight dynamic models that simulate each of flying objects in an

isolated space and therefore have no mid-air collision estimation capability, in this implementation, not even it can estimate multiple objects dynamic simultaneously, but also it is possible to simulate their collision and impact reactions. Therefore, we can simulate every aspect of a cooperative flight mission in the future. For example, Figure 4.6 shows two drones colliding with each other while landing on the platform (ground colliders).

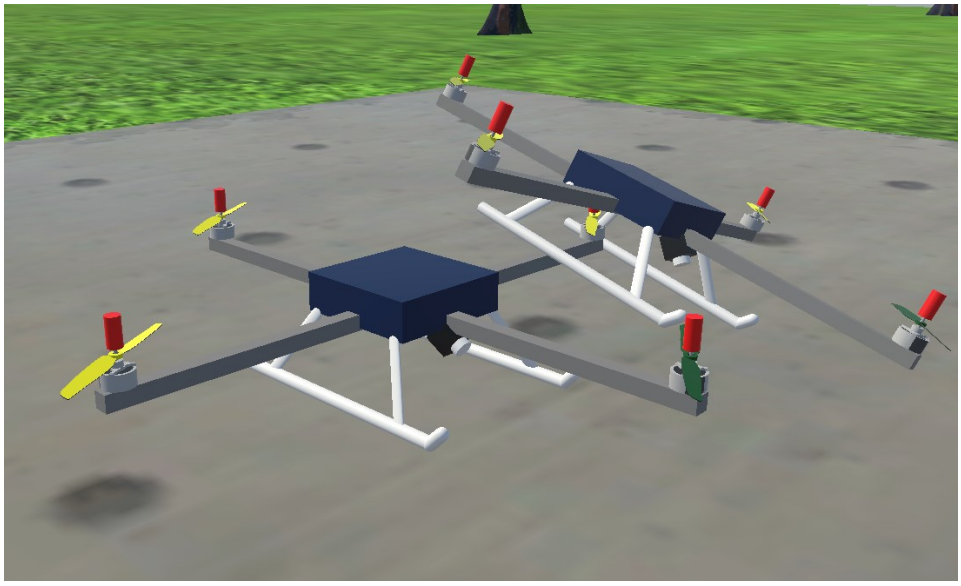


Figure 4.6: An example of two drones colliding with each other

4.3 Sensor simulation

The plant outputs the system state that is not necessarily similar to the desired feedback. For example, if the autopilot anticipates the system position in the spherical coordinate system using latitude, longitude, and elevation (like what it gets from the actual GPS module) and at the same time, the plant calculates it in earth-centered-earth-fixed (ECEF) Cartesian coordinates in 3-space [33]; it has to be converted in order to close the loop properly. A group of functions is needed to convert them to the system feedback. Besides the output format, the sensor refresh rates need to be the same. For example, the actual GPS sensors refresh rate is 10 Hz, while the simulator can calculate it more than 1000 Hz. A refresh rate adapter needs to be implemented because there is less or no limitation on the simulation rate (just limited by the hardware capacity). The refresh rate adapter just drops the data till the next time slot for the sensor output. For example, if it is intended to drop a 1000Hz data rate to a 10Hz, it drops 99 data frames for every single data that it sends.

4.3.1 Fire, thermal camera, and computer vision simulation

To simulate a fire detection system, you need to implement fire in your simulation environment. Since we do not want to research or investigate the fire behavior itself, it is chosen to simulate as much as needed to reduce processing power. If we tried to simulate the fire behavior ultimately, it requires much bigger processing power, and most of the CPU and GPU usage would have been occupied by that task and not the HIL simulation. Given that any algorithm needs to be run in real-time and any other non-real-time algorithms (even the accurate ones) are not viable solutions for HIL simulation, any possible yet decent form of implementation may seem acceptable. Therefore, the fire system is modeled as an animation captured from a real fire. This fire system is provided by Unity as part of their particle system [34]. We have total control over the fire behaviors; namely, the fire position can be random or programmed, it can grow or diminish, and it emits smoke or not. A random fire position helps us to find any possible corner or edge cases in our algorithms. Now we needed something to detect the fire and distinguish over forest environment.

Since one of the goals is to test our actual computer vision algorithms, the simulated camera output has to be kept similar to the original format of the real camera on the platform. The image data structure and the pixel format are similar to a greyscale image. There are several different image formats for thermal cameras (Shown in Table 4.1). We have chosen the White Hot format because it was easier to process and has a smaller image size. It displays the warmer objects brighter and cooler zones darker. In our case, it has an 8-bit format (a 0-255 integer for each pixel). In this image format, the absolute black is 0, and the absolute white is 255. To test some basic computer vision algorithms such as threshold or the object centroids and detect the fire coordinations according to the corresponding pixels extracted from the thermal camera, we send the simulated camera image to the flight controller to check if the image processing algorithms run efficiently enough on the autopilot microcontroller and do not obstruct its critical tasks such as stabilizing and navigation. The result and the implementation of this function are described in the case study section of this thesis.

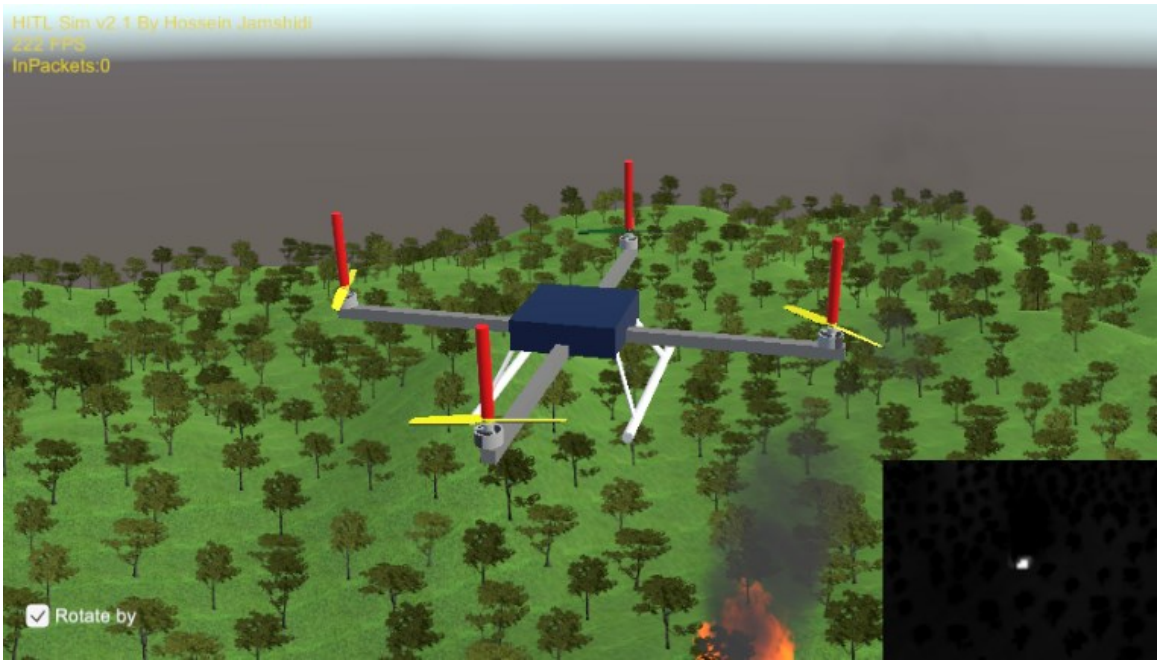


Figure 4.7: The thermal camera view of drone

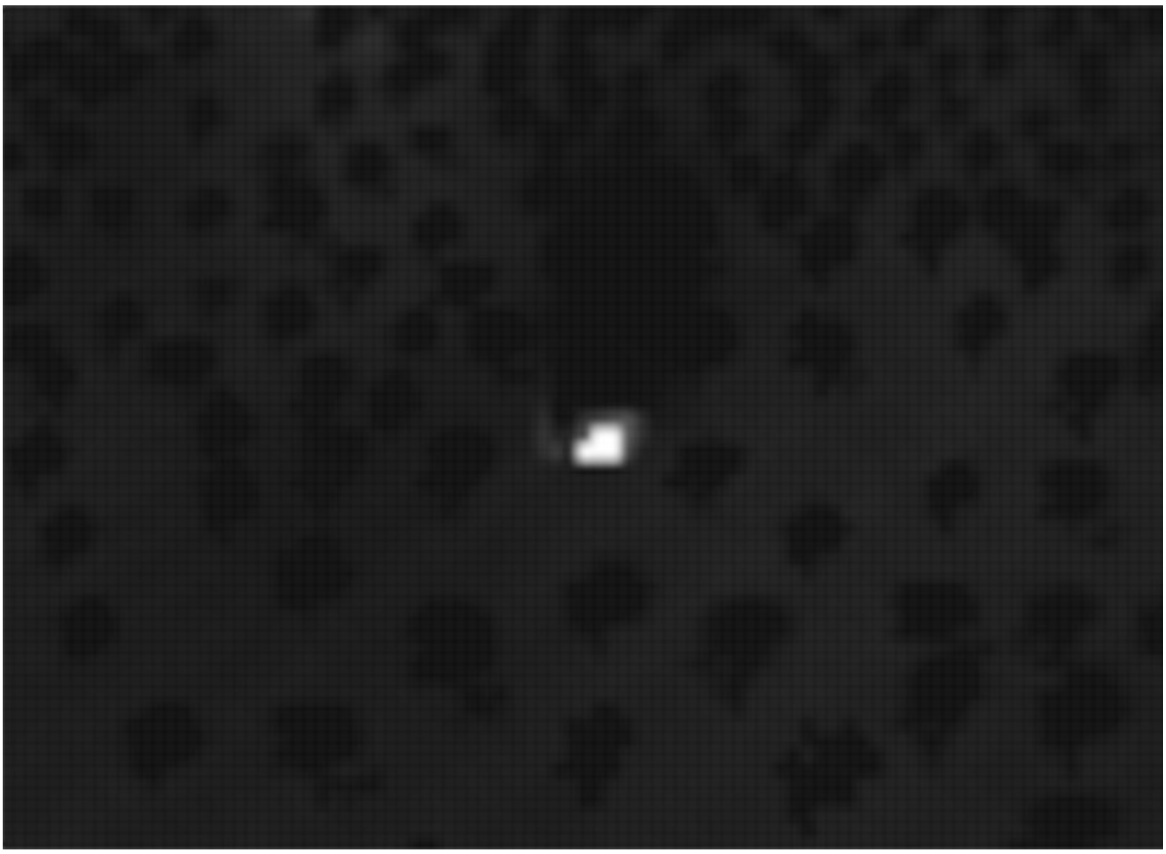
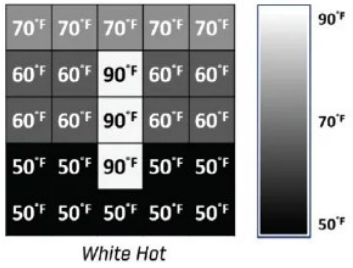
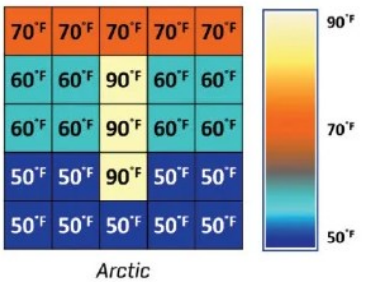
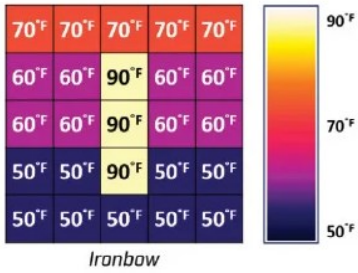


Figure 4.8: Simulated thermal camera image

Table 4.1: Thermal image formats

Format	Color pallet	Image sample
White Hot		
Arctic		
Iron Bow		

4.3.2 Adding noise and induced fault

To challenge the system controller and make it more realistic, we can add white noise to the data before sending it to the autopilot. The noise model that we used is a white Gaussian noise implanted as a part of the GCS HIL interface and controlled by the user in real-time. Besides, the function that sends the data can make faulty data too. For example, you can cut a sensor data stream as if it was damaged. This particular part helps us to design and implement fault-tolerant control algorithms for the subjected system. Also, there is a normally disabled ballast weight on each drone that could be enabled in order to make the drone weight distribution slightly unbalanced in order to test the controller behavior.

4.4 Summary

In this chapter, the implementation of the simulator is elaborated. In the beginning, the basic idea behind this type of simulation is discussed, and the reasons why it is needed are also described. Second, the different part of the simulation is introduced. In the next section, the designed 3D environment is discussed, and the reason why Unity Engine was selected to perform the low-level graphical tasks is elaborated. The next section describes the flight dynamic model that is used for the simulation and talks about the 6DOF and collision solver. The last section is about the sensors' simulation principles. The methods used for the visualisation of the fire is elaborated. Furthermore, the method for simulating a thermal camera image is discussed. In the end, the system disturbances such as noise and sensor failure are discussed.

Chapter 5

Interfaces

5.1 The interface explanation

This unmanned system has three separate components: the autopilot, GCS, and the simulator. Unlike the autopilot that is a physically separate device, the GCS and the simulator are software running on the same computer. These three components should exchange data bi-directional instantly. Time delay is a critical matter here. Any measure that leads to eliminating communication delay has to be taken into account and investigated. Compressing the data and making them as shorter as possible alongside multi-threading implementations are our main focus and most viable solutions to reduce the communication delay time between components.

As the relation of the components has been depicted in Figure 5.1, the GCS uses the user datagram protocol (UDP) [26] port to exchange data with the simulator. On the other side, the GCS utilizes a high-speed serial port to the autopilot hardware and also another serial port with regular speed to transfer data during flight using the RF modem. In the following sections, both types of communication are discussed.

Figure 5.1: The components' connection diagram

5.2 Simulator to GCS interface

The simulator and the GCS are both windows applications running under the same computer at the same time. These two applications need to communicate and transfer data bi-directionally (Figure 5.2). There are several well-known and confirmed methods for communication technics to serve the purpose in this circumstance: UDP port and shared memory.

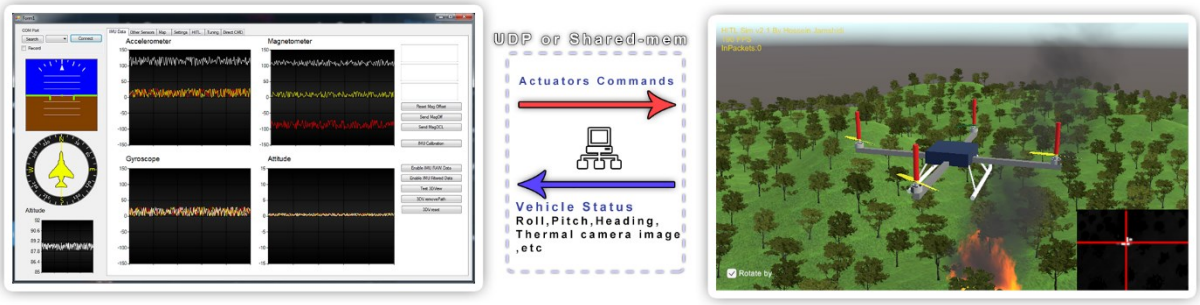


Figure 5.2: GCS to Simulator interfacing

5.2.1 UDP Port

In computer networking, the UDP is one of the main members of the Internet protocol (or IP) suite. With UDP, computer applications, whether in the same device or over a network, can transfer data, in this case, referred to as datagrams, to other applications on the same network address or IP. Unlike some other means of network communication, Prior communications are not necessary for setting up the communication.

UDP uses a simple connectionless communication model with a minimum of protocol mechanisms that keep the interface fast and eliminate unnecessary latencies. UDP provides checksums for data integrity to keep the data reliable, especially for critical purposes at the lower level.

It also uses port numbers for addressing different functions at the source and destination of the datagram. For example, the user can set up different pip lines of data to the same address with the different port numbers that could be anything from 0-65536 (if it was not occupied already), and there would be no data conflict.

Like any other method that has its own pros and cons, UDP has its own downside too. It has no handshaking dialogues, thus exposes the user's program to any unreliability of the network layer. For example, if a data packet fails, there is no retry mechanism at the lower level or any notification; it is just like the serial port; it sends and forgets the data. If the data transfer system needs more reliability, the protection mechanism has to be implemented at a higher level by the user or consider a TCP/IP communication that has more delivery protection. However, the TCP/IP is not suitable for this purpose since it has a much higher latency than the UPD port.

Table 5.1: UDP datagram header

Offsets	Octet	0								1								2								3			
Octet	Bit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	Source port																Destination port														
4	32	Length																Checksum														

Table 5.1 shows the UDP datagram header structure. A UDP datagram has a datagram header (4 Bytes) and a data section that is the user data. The UDP datagram header consists of 4 fields, each of which is 16 bits (2 bytes). The user application data follows the header.

Source port number shows the sender's port, and it is possible to use it if the other side waits for an answer. It is possible to leave it zero if not applicable. The destination port number contains the receiver's port, and it is necessary; it tells the destination address. Imagine a letter or a parcel; if it does not have the sender's address, it would be a dead end. The length determines the

length of both the UDP header and UDP data. The checksum field that its principals and basics are described in its section could be used for the error protection mechanism of the data.

5.2.1.1 UDP cummincation implementation in C#

Since both the simulator and the GCS are written in C#, we describe a simple UDP communication in C#. The communication structure contains a listener and a sender. The listener is called the server, and the sender is called the client. In the example, the server listens on the 2021 port, and if it gets any data on the port, it sends a packet with a constant string containing "received" work as the acknowledgment. It stores the data in RecvBuffer for further process or parsing.

```
UdpClient udpServer = new UdpClient(2021);
IPEndPoint remoteAddress = new IPEndPoint(IPAddress.Any, 2021);
Byte[] ByteReply = ASCII.GetBytes("Received!");
while (true)
{
    Byte[] RecvBuffer = udpServer.Receive(ref remoteAddress);
    Console.Write("The data is receive from " + remoteAddress.ToString());
    udpServer.Send(ByteReply, ByteReply.Length, remoteAddress); // reply back
}
```

Now a sender is needed on the other side to communicate with the server. Since both applications (GCS and simulator) are running under the same computer, the server's address should be 127.0.0.1, which is the current localhost's address.

```
Byte[] ByteToSend = ASCII.GetBytes("Data from the client!");
UdpClient udpSender = new UdpClient();
IPEndPoint serverAddress = new IPEndPoint(IPAddress.Parse("127.0.0.1"), 2021);
// 127.0.0.1 is the server ip address.
client.Connect(serverAddress);
client.Send(ByteToSend, ByteToSend.Length);
Byte[] receivedData = client.Receive(ref serverAddress);
// receivedData has to be equal to "Received!" if everythings goes well.
```

```
Console.Write("reply data from server " + serverAddress.ToString());
```

This snippet code sends sample data to the server and waits for the replay. When the data is ready to be transferred, it has to be translated into a data packet using a convention called the protocol. The protocol is elaborated in the protocol section.

5.2.2 Shared memory

Shared memory is another method for inter-process data transferring. In computer science, shared memory is a memory that may be concurrently used by multiple applications to provide data communication among them. Shared memory is one of the most efficient methods to pass data between independent processes. In windows, shared memory uses a method called Memory-Mapped Files. A Memory-Mapped Files maps the contents of a file to the memory. There two types of Memory-Mapped Files: Persisted and non- Persisted. Persisted Memory-Mapped Files make a memory-mapped object from an actual file on the physical disk; the file is readable by the user and has an accessible address. On the other hand, the non- persisted method does not create any file on disk thus is more desirable for the purpose of this research.

5.2.2.1 Shared memory implementation in C#

In this section, the method is briefly explained using a simple example. Imagine there is a program called DataWriter that writes the data to the shared memory and another program caller ProgramReader; the implementation could be similar to this part of the code:

The ProgramWriter:

```
using system;
using System.IO.MemoryMappedFiles;

class DataWriter
{
    static void Main()
    {
        MemoryMappedFile mmfObj =
            MemoryMappedFile.CreateNew("ThesisTest", 1000);  // 1000 is the
      size of the object
        MemoryMappedViewAccessor accessorObj = mmfObj.CreateViewAccessor();
        for (int i =0;i<100;i++){
```

```
            accessorObj.Write(i, 0x41); // 0x41 = 65 = A
        }
        Console.ReadLine(); // Wait for a keyboard input.
        accessorObj.Dispose();
        mmfObj.Dispose();
    }
}
```

This program creates a non-persisted shared memory object and writes the ASCII character "A" or binary value of 0x41 to the memory and repeats the same thing 100 times for 100 consecutive addresses. Now we need another process to read the data.

The ProgramReader:

```
using System;
using System.IO.MemoryMappedFiles;
class ProgramReader
{
    static void Main()
    {
        MemoryMappedFile mmfObj =
            MemoryMappedFile.OpenExisting("ThesisTest");
        MemoryMappedViewAccessor accessorObj = mmfObj.CreateViewAccessor();
        int tmp =0;
        for (int i =0;i<100;i++){
            tmp = accessorObj.ReadByte(i);
            Console.WriteLine("The value in {0} is {1}",i, value);
        }
        accessorObj.Dispose();
        mmfObj.Dispose();
    }
}
```

This program accesses a non-persisted shared memory and reads the ASCII character in the memory, and prints it in the console window. If everything goes well, it should read the same 100 "A" characters.

These examples were simplified for describing the method. A decent implementation has to use a lock mechanism to prevent processes from interfering with each other. The lock mechanism technique is out of the scope of this thesis [35].

In the section 3, the two methods were compared, and the reason for choosing UDP over shared memory was explained. Also, in section 3, multi-threading was briefly elaborated, and its necessity for this research was discussed.

5.3 GCS to autopilot interface

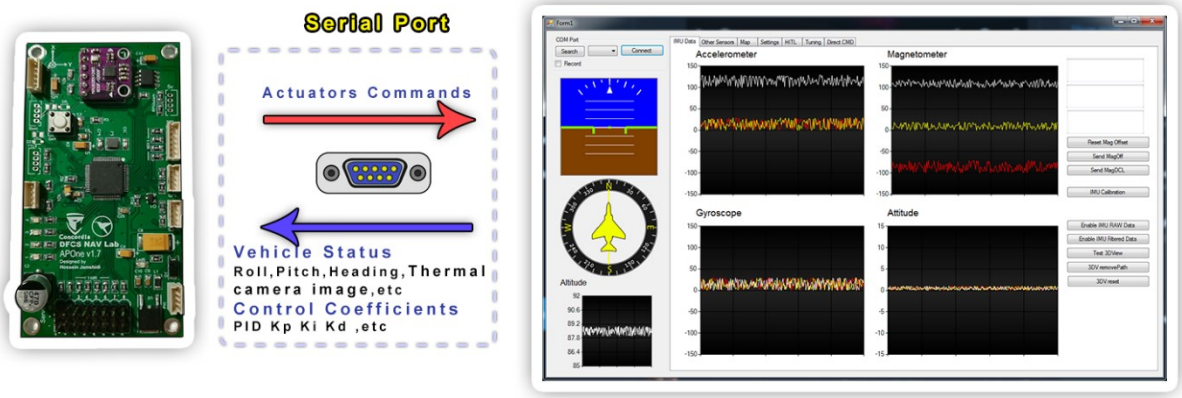


Figure 5.3: GCS to Autopilot interfacing in HIL mode

In section 3, the method for communicating with autopilot at the physical layer was briefly explained. In this section, the concepts and the principles are elaborated. There are two serial ports linking autopilot to the GCS: The telemetry link and the HIL link.

5.3.1 The telemetry link

The telemetry link is the ordinary serial port with a medium baud rate (57,600bps). This serial port could be coupled with an RF modem (Figure 5.4). For RF modems, when the Air baud rate increases, the link quality decreases. In other words, higher data rates cause more data loss, and the maximum range of the wireless link gets decreased; thus, a trade-off between range and data rate has to be chosen.

Figure 5.4: Wireless data module

To optimize the telemetry data rate, an optimized refresh rate must be considered. For example, the GPS module sends 10Hz of positioning data to the flight controller; even though it is necessary for the navigation controller, only 1Hz of that data is needed on the ground since it just has monitoring purposes. For another instance, the flight controller estimates the roll, pitch, and yaw 1,000 times per second (while it is even impossible to send this big data size via a wireless link) for the ground control operator, even 10Hz of attitude data is enough.

5.3.1.1 Half-duplex radio link

Another important aspect of wireless telemetry is that almost all inexpensive data modules have a half-duplex interface. Unlike the cable data transfer that is full-duplex, a half-duplex interface can only send or receive at a time. In other words, while it sends data, it couldn't receive any data and vice versa.

The strategy to tackle these issues is to send and receive data in an ask and response fashion. One of the sides – GCS here – must be chosen as the master or the moderator. The moderator asks the other side for data and waits for the answer. The slave does not send any data unless was ask. For example, if the autopilot wants to send attitude data to the GCS and the GCS decides to send config data simultaneously, one of them would be lost; the solution is to let the GCS ask for attitude data if needed and do not ask for it when it needs to send data to the autopilot. In other words, the GCS decides when autopilot sends data to make sure it does not lose any data.

5.3.2 HIL link

The GCS has another serial port connected to the flight controller, the HIL link. The HIL link is only used during HIL simulation and has no usage during a normal flight. Since the HIL serial port, unlike the telemetry link, uses a cable and also HIL simulation would be done under a controlled lab situation, it is less likely to get interference from surroundings; thus, it is possible to use higher baud rates (even up to 921,600 bps). Because it is a full-duplex communication does not need to ask-and-response schema and can send and receive simultaneously.

The data packet rate between FDM and the GCS is 500Hz using a data frame that contains all the data we need to for the HIL. We have to keep in mind that the effect of the data refresh rate is significant, too; it has to be closer to the actual situation. If the data or control loop of the simulation and real-world situation has a significant difference, the result might be useless. For example, for the positioning data from the GPS module, we have a 10Hz-18Hz maximum of that data; if we send 500Hz of positioning data to the autopilot in the simulation, our simulation result won't be accurate at all and may show some false good results. To address this requirement, we send a 500Hz chunk of data, including simulated sensor data such as roll, pitch, and heading, plus a 10Hz of other data such as latitude, longitude, altitude, speeds, etc. that is closer to our actual sensors.

5.3.3 Data compression

One of the other methods to reduce latency is to keep the data frames as shorter as possible. But sometimes, the data frame has to be bigger than a certain size. For example, if the autopilot should send a floating-point number like [-145.264862060546875], it takes 20 characters/bytes; if any of those numbers get truncated, the accuracy decreases. To reduce the size without losing the accuracy, a lossless compression method must be used. To use a binary protocol to compressed the data frame as much as possible is one of the simplest yet efficient techniques. The binary protocol uses the same format that the compiler uses to store the data. For example, if you want to send a floating-point number like the latitude and longitude in the uncompressed format, it takes 20 bytes in an average case for each, while the same data in binary form only needs 4 bytes in all cases (Table 5.2).

Table 5.2: Text and binary format comparison

Format	Data	Lenght
Text	-145.264862060546875	20 bytes (average cases)
Binary	0x[C3 11 43 CE]	4 bytes (all cases)

5.3.4 Protocols

The protocols are our convention of data exchange. It determines the data structures. Unlike packet or payload base communications, the serial port does not have any built-in frame structure; the smallest unit in the serial port is a byte. The serial port works as a data stream. It sends the data frame byte by byte. A method is needed to determine the beginning of each frame. The implemented protocol has preambles, receiver ID, message ID, Data payload, and the checksum (Table 5.*3*).

Table 5.3: GCS to Autopilot data transfer protocol

	Preamble*	Receiver ID	Message ID	Data length	Data	Checksum
Lenght	3 Bytes	1 Byte	1 Byte	1 Byte	= Data length	1 Byte

*Preamble is always 0x7096AC

5.3.4.1 Preamble

For this thesis research, a three bytes preamble method is used. Each frame of data has the same preamble; here, it is **0x70**, **0x96**, and **0xAC**. A safe preamble has three different bytes. The listener reads the streams until it gets the three bytes in the right order. After that, it starts to deal with the rest of the data as the message body. In some cases that the protocol length is not constant and does not have any length indicator, a closure is needed too. Closure works the same as preamble but is at the end of transmission and must be different from the preamble.

5.3.4.2 Receiver ID

The next byte after the preamble is the receiver ID. Receiver ID is an 8-bit (0-255) indicator that determines whether the data belongs to the current receiver or not. To have multiple aerial systems or, in general, multiple agents, a receiver ID has to be explicitly assigned to all components since all the system members are getting the data on the same wireless link on the same frequency simultaneously. The master system, the GCS, always has zero receiver ID. For example, if a system

has one GCS and three aerial systems, the GCS ID is zero, and the rest are 1, 2, and 3; if GCS wants to send data to the #3, it sends the data frame with the receiver id equal to 3. As mentioned earlier, all the other receivers get the message, but #1 and #2 drop it. The #3 runs it and sends the result (or the acknowledge message) to the GCS with a receiver ID of zero that all other units ignore.

5.3.4.3 Message ID

The next byte after the receiver ID is the message ID. Each data frame has an 8-bit message ID. The message ID determines the frame data structure. For example, the message ID of 0x1D belongs to attitude data. After parsing the message ID, A state machine decides how to decode and verify the data. The data frame format is shown in Table 5.*3*.

5.3.4.4 Data payload

The data payload contains a series of bytes (the data body) and a length indicator. Since each frame has its length and is not constant, a length indicator is needed. The length indicator is an 8-bit integer (0-255) that determines the data body length and comes at the beginning of the data payload. For example, if the data body contains n bytes, the data payload is:

Table 5.4 Data Payload

	Length indicator	1	2	….	n-1	n
Data Payload	n	D_0	D_1	….	D_{n-2}	D_{n-1}

Keep in mind, n can't be more significant than 255; thus, the data body length must be less than 255.

5.3.4.5 Checksum

A critical part of the implementation is to verify the data on each side. A ubiquitous method to fulfill this requirement is to add a checksum, a small attached data calculated from the sending data before sending. When a unit (GCS or Autopilot) receives the message, it re-calculates the checksum from the received data and compares it with the received checksum; if it was the same, it releases the data for parsing in the data refiner; otherwise, it drops it. For example, if the checksum operator is summing all the data and the data frame contains six bytes of some sample data, in that case, it has: (Table 5.5)

Table 5.5: checksum calculation

	D_1	D_2	D_3	D_4	D_5	D_6	Checksum
Data	10	11	12	13	14	15	10+11+12+13+14+15 = **75**

Now let say it got a defective data frame due to a bad data link (D_3 should have been 12, but it got zero)

Table 5.6: Wrong checksum sample

	D_1	D_2	D_3	D_4	D_5	D_6	Received checksum	Calculated Checksum
Data	10	11	0	13	14	15	**75**	10+11+0+13+14+15 = **63 ≠ 75**

Since the received checksum (75) is not the same as the calculated checksum (63), it drops the frame.

In a real case, the checksum operators are sophisticated algorithms like CRC32, CRC16, or Fletcher-32. The most common algorithms are 32bits (4 Bytes), but since we needed the shorter data length and calculation speed, we have to sacrifice performance over speed. Our checksum operator is to XOR all of the bytes in the data frame. XOR shows a significant performance while compering with its easiness. It is just 8-bit. While XOR operator is enough for our hardware-in-the-loop simulation purposes, for a real-world application, it's better to implement something more reliable like CRC32 or even CRC64, or if it is supercritical or has a long data frame, consider SHA256 algorithm.

5.3.4.6 Encryption

While using a wireless connection, the data signals do not go just to a receiver; it transmits to space omnidirectionally. It opens a loophole in the system that allows other unauthorized stations to access the data, read the telemetry data or, in a more harmful way, get control of the platform from the operator by hacking the protocols. Encryption is a method to keep the system data safe and secure. It can make the data unreadable for unauthorized receivers. For example, 256-bit AES is one of the widely used techniques. The Advanced Encryption Standard (AES) is a standard for encrypting electronic data announced by the U.S. National Institute of Standards and Technology (NIST) in 2001 [36]. The implementation of 256-bit AES is part of the future work of this thesis since all the research phases have been done in a controlled area but are part of future implementation.

5.4 Summary

In this chapter, the interfacing between the simulator and the GCS and GCS to autopilot hardware was elaborated. Different inter-application and inter-process communication methods were discussed, and the implementations were briefly described. The performance of the methods was compared, and the reason why the UDP port was chosen is explained. Furthermore, the wireless telemetry link characteristics were described. RF modem half-duplexing issues were explained, and the solution was discussed. To fullfile this chapter, the data protocols were explained. The methods used for sending the payload data, identifying the receiver, verifying the correctness, and encrypt the data were discussed.

Chapter 6

Autopilot Firmware

6.1 Firmware definition

In computer science, firmware is a specific type of computer software that performs the low-level regulation for a device's specific hardware. Imagine the firmware as software that runs on embedded hardware and does not need a desktop PC or any other desktop-level frameworks. Firmware (such as the microcontroller of a robot, washing machines, or a smart fridge) may contain only essential functions of a device and may only deliver services to higher-level software; in this case, an autopilot has to have a highly sophisticated structure. It performs the control algorithm, communication tasks, data acquisition functions (reading sensors), state estimation, and etc. [37][38].

To design the firmware, especially for such a critical application, the documents from the manufacturer must be studied carefully. Usually, some data about the registers and peripherals have a very high level of granularity and need a considerable background knowledge of the topic. Elaborating on the full implementation of the firmware is out of the scope of this thesis and may include more computer science material that is not the goal of this thesis; therefore, it is elaborated in the three most important levels: choosing the right integrated development environment (IDE) and the debugger, implementation of the communication tasks, implementation of the controller, and data acquisition.

6.2 IDE and debugger

The first thing you need to write your firmware is the correct IDE and debugger. Usually, the chip manufacturer provides the options, and you have to choose among them. In this case that a STM32F7 series MCU is used, there are several major options for the IDE, including MDK ARM, IAR Embedded Workbench, Arduino Pro IDE, and the STM32CubeIDE. The cost of each IDE is described in Table 6.1.

Table 6.1: The avalibale IDEs for STM32s

IDE name	Supplier	Cost
Keil MDK Arm	Keil	4200 CAD/Year
IAR Embedded Workbench	IAR	5995 CAD
Arduino Pro IDE	Arduino	Free
STM32CubeIDE	ST	Free

Even though the Arm Keil MDK and IAR Embedded Workbench have astonishing performance and debugging toolsets, their licensing cost is not affordable for this research. Therefore the next two options seem more rational. Arduino Pro IDE is a lightweight software and has a good online community; however, it does not support the professional debugging capabilities that the STM32CubeIDE provides; thus, STM32CubeIDE has been chosen for the purpose of this research.

The next thing that needed to be chosen is the debugger. The debugger is hardware that connects and flashes the microcontroller. The debugger is a bridge between IDE software and the microcontroller. It helps to see the microcontroller's behavior in real-time; It helps you to monitor the registers and variables. STMicroelectronics, the MCU manufacturer, recommends to use ST-link series debuggers that have a good performance and is very affordable; it costs 45CAD. Figure 6.1 shows the ST-Link V2 Debugger.

Figure 6.1: ST-Link V2

There are several ways to connect and debug the microcontroller. For the purpose of this research, the serial wire debugging (SWD) port is used because it needs less wiring comparing with JTAG or other debugging methods. It just needs a three-wire connection. Figure 6.2 shows the SWD port schematics that have been designed into APOne v1.7 hardware.

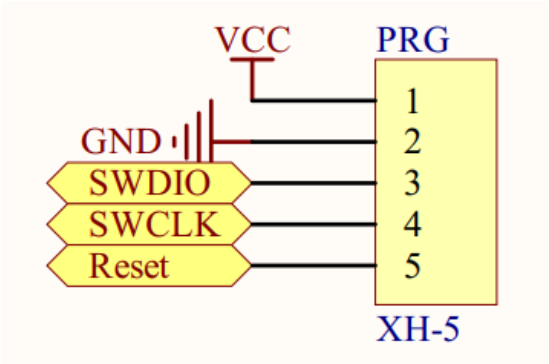


Figure 6.2: SWD connection

6.3 Communication

For communicating with the GCS and HIL two UART port is used as it was described in Chapter 5, one for GCS wireless link and another one dedicated to HIL simulation. In this section, the protocol implementation is elaborated and used as a sample to fulfill the task. Using the UART port on the MCU at the peripheral and register level has a high degree of complexity that is skipped in order to emphasize the more important aspects.

The programming language the is used for the firmware is C++. One of the important behavior of C++ that needs to be discussed before describing the data conversions is bitwise mathematics and bitshift operators.

6.4.1 Handling bitwise operators in C++

There are 6 different bitwise operators in C++: AND, OR, XOR, Left Shift, Right Shift, NOT show the symbols.

Table 6.2: Bitwise operators

Operator	Description	Example
&	Bitwise AND: The result of AND is 1 only if both bits are 1	a&b gives 1
|	Bitwise OR: The result of OR is 1 if any of the two bits is 1	a|b gives 13
^	Bitwise XOR: The result of XOR is 1 if the two bits are different	a^b gives 12
<<	Left shift: In C or C++ takes two numbers; left shifts the bits of the first operand, the second operand decides the number of places to shift.	b<<1 gives 18
>>	Right shift: In C or C++ takes two numbers; right shifts the bits of the first operand, the second operand decides the number of places to shift.	b>>1 gives 4
~	Bitwise NOT: Takes one number and inverts all bits of it	~a gives 250

- In all the examples, a is 5 and b is 9

6.4.2 Data handling

The output of the autopilot data parser is a byte array. A byte array is a series of bytes that needs to be converted to meaningful data. To convert the byte array, two methods can be used: the bitshift method and the memcpy method.

6.4.2.1 Bitshift approach

The bitshift technic contains some manual data handling. An example of bitshift technique to construct a 32-bit integer from a 4-byte array is presented below:

```
a  = (bytesIn[0] & 0xFFFFFFFF); // No shift
a |= (bytesIn[1] & 0xFFFFFFFF) << 8 ;
a |= (bytesIn[2] & 0xFFFFFFFF) << 16 ;
a |= (bytesIn[3] & 0xFFFFFFFF) << 24 ;
```

It may seem simple, but it gets too complicated and error-prone when the structure gets more sophisticated.

6.4.2.2 Memcpy approach

In this approach, it uses memcpy function to cast the data to the variable. No knowledge of data type is needed. The data and the destination type have to be the same. It is simple and fast. An example of memcpy technique to construct a 32-bit integer from a 4-byte array is presented below:

```
int a =0;
memcpy (bytesIn, &a, sizeof(a));
```

A simple usage of this method is to convert the transferred HIL data to flight data of the autopilot. Imagine the flight data as a C++ struct defined as follow:

```
typedef struct _flightdata
{
    float roll;
    float pitch;
    float heading;
    float Lat;
    float Lon;
    float Baro_Alt;
    float GPS_Alt;
    float GSpeed;
    float ASpeed;
    float SidS;
    float AoA;
    float ax;
    float ay;
    float az;
    float p;
    float q;
    float r;
    int WPIndex;
    int WPtotal;
}FlightData;
```

To convert it using the bitwise operators takes too much effort and may produce some errors. But with the memcpy technique, it will be easy and error-proof as far as the data format is correct. A simplified converter would be something like this:

```
FlightData fdata;

void calcHIL(uint8_t bytesIn[],uint8_t frmlenght)
{
    if (frmlenght<sizeof(fdata)){
        return;
```

```
    }
    memcpy(bytesIn, &fdata, sizeof(fdata));
}
```

Both these approaches are useful and have to be used with caution. None of the above approaches are reliable unless a complete unit test verifies them. Finally, a good data set for the tests need to be prepared and be used to verify the method implementations.

6.5 C++ controller implementation

The controller that is implemented in the autopilot is a PID controller. The PID controller contains three controllers: P or proportionate, I or Integrator, D or derivative. Each part has its own gain, too; the gains are named Kp, Ki, and Kd. The structure of a simple PID controller is depicted in Figure 6.3.

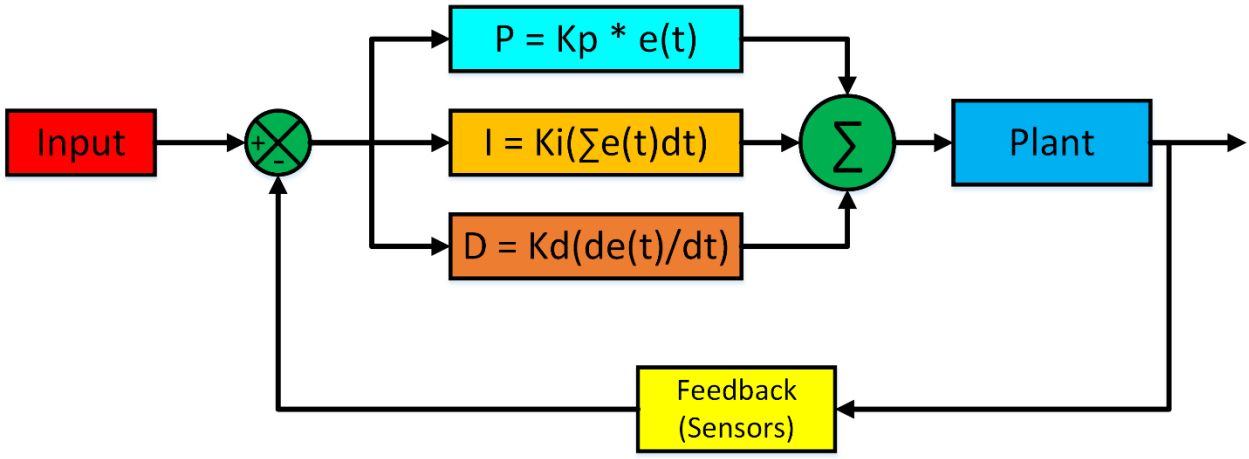


Figure 6.3 PID controller

In order to use it in the real embedded system, this concept needs to be converted to a digital version in C++. In the span of this research, a new implementation of PID controller is designed to suit the application. Therefore, the implementation was the sole PID controller used during the simulation and the real-world test, which is confirmed via experiments. The test results are discussed in Chapter 7. The PID controller is designed into a class object to keep the implementation portability and follow the C++ objective design patterns. The full implementation source code is attached in Appendix A. Here, the usage is briefly discussed. The first thing that is needed to use a class object is to declare the object. The class name is PID; therefore, the definition of the object will be something similar to this part of the code:

```
PID rollControl;
```

"rollControl" is our controller object. Now, it needs to be initialized. Initializing a PID controller includes the gain assignment and setting the range. For example:

```
rollControl.reset();
rollControl.setK(3.00f, 0.5f, 0.2f); // setting the gains(Kp, Ki, Kd)
rollControl.set_ITerm_Limits(-50f, 50f); // Clamping the Iterm (Min,Max)
rollControl.set_output_Limits(-500, +500); // Clamping the output (Min,Max)
```

Now, it can be used in the control loop by calling the "updatePID" method. This method has two arguments; the first one is the sensor reading, and the second one is the desired set-point. For example:

```
rollOutput = rollControl.updatePID(roll, desiredRoll);
```

This implementation can be expanded and used for any type of controller; For example, it can be used for pitch angle, heading, navigation, or altitude. This controller is the smallest cell of the control system. The more important point that needs to spend more time on it is the controller structure. The controller structure means the order of controllers. For example, imagine a fixed-wing airplane; if a controller is designed to control the pitch angle of the platform, another controller needs to be designed in order to set the proper pitch angle to achieve the desired height; and also another one needs to airspeed in order to prevent the platform from stalling. In section 7.2, the controller structure of the platform used for the case study is elaborated.

6.6 Data acquisition

The process of gathering data from the environment is called data acquisition. For a control system like an autopilot, the data comes from sensors. Sensors could include accelerometers, gyroscopes, laser altimeter, battery voltage, etc. Each sensor has its own method of data acquisition. Some of the common methods for communicating with modern digital sensors include SPI, I2C, CAN. CAN bus is more used in automotive industries; thus is out of the scope of this research [39]. The SPI or Serial Peripheral Interface is a synchronous serial communication interface used for short-distance communication, mainly in embedded systems. SPI devices transfer the data in full-duplex mode using a master-slave arrangement with one side as the master and the others as slaves. The master device creates the data frame for reading and writing. Multiple slave devices are supported through a selection of an individual slave select pin (SS) or chip select (CS) lines. Figure 6.4 depicts the SPI bus diagram for one master and multiple slaves [17].

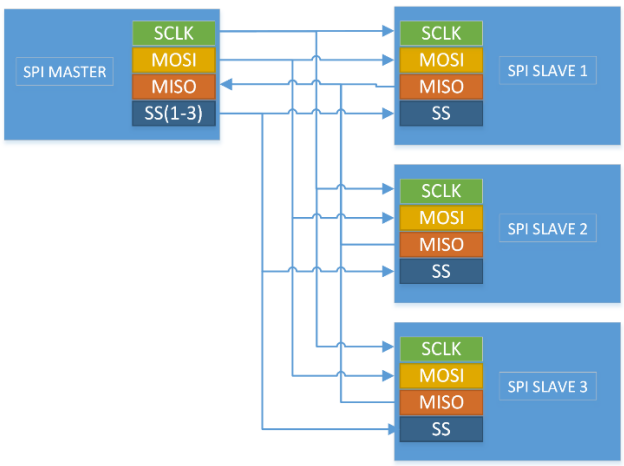


Figure 6.4 SPI BUS Block diagram

On the other hand, I2C (Inter-Integrated Circuit) is a synchronous, multi-master, multi-slave, packet-switched, single-ended, serial communication bus. It is widely used for attaching lower-speed peripheral ICs to processors and microcontrollers in short-distance, intra-board communication [40].

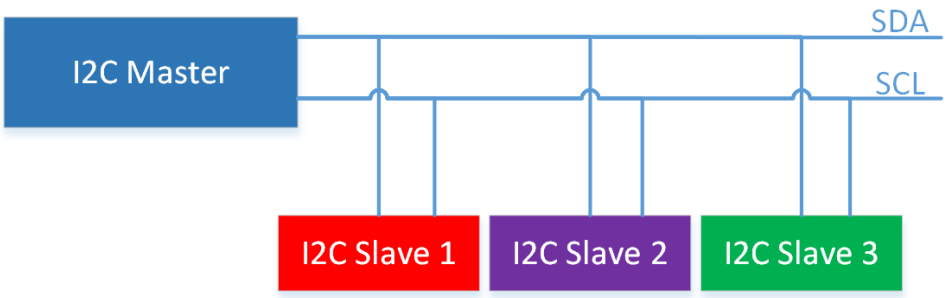


Figure 6.5 I2C bus diagram

Unlick the SPI bus that selects slaves using CS pin, the I2C bus organize the slave by their addresses. Each I2C slave device has a unique 7-bit (0-127) address that the master device uses to invoke the specific slave device. There are several I2C implementations for an ARM Cortex based MCU (the MCU that is used for this research). A straight forward implementation has been chosen for the firmware. The Reader function is as follows:

```
uint8_t I2C_Read(I2C_HandleTypeDef* Handle, uint8_t device_address, uint8_t
register_address, uint8_t* data) {
    if (HAL_I2C_Master_Transmit(Handle, (uint16_t)device_address,
&register_address, 1, 100) != HAL_OK) {
        if (HAL_I2C_GetError(Handle) != HAL_I2C_ERROR_AF) {
        }
```

```
        // Return error
        return 0;
    }
    // Receive bytes
    if (HAL_I2C_Master_Receive(Handle, device_address, data, 1, 100) !=
HAL_OK) {
        // Return error
        return 0;
    }
    // Return OK
    return 1;
}
```

For example, to read the register at `0x75` of the MPU-9250 device (address is set as `0xD0` by the manufacturer):

```
uint8_t MPU9250_WHOAMI(I2C_HandleTypeDef* Handle) {
    uint8_t read=0;
    if (HJ_I2C_Read(Handle, 0xD0, 0x75, &read) != 1) {
        /* Return error */
        return 0; // Should return 0x71
    }
    return read;
}
```

6.7 Summary

In this section, the system firmware was defined and characterized. In the next part, different IDEs for developing the firmware were discussed, and the reason why STM32CubeIDE has been chosen was elaborated. The debugger was expanded, and its characteristics were pointed out. The next part was dedicated to autopilot communications; mainly, the different methods for the protocol parsers were compared and clarified. In the end, the system data acquisition was discussed, and main methods such as SPI and I2C were elaborate; a code snippet is used to clarify the methods.

Chapter 7

7 Case Study

7.1 Test definition

To confirm our methods, a simple test is designed to test the complete system's consistency and functionality. It starts with a takeoff and then moving toward the fire and using an algorithm to detect and report the fire coordination. Even though the main goal for this case study does not include any sophisticated search algorithm test, and it is meant to emphasize the system performance, it is a good starting point for the further development of advanced searching and scanning algorithms.

7.2 Controller design

To perform the search mission, the first step is to make the platform stabilized and controllable. The stabilization task is the most time-critical part of the loop since it has the highest running frequency. The method that is used for stabilizing the platform is a well-tuned PID controller [41][42]. We tried to keep it platform-independent to be able to use it on embedded and non-embedded applications. PID gains are tuned manually by observing the system's behavior. Because the goal for this task was to test the system consistency and make the platform fly-worthy, it didn't emphasize PID tuning technics (such as an adaptive controller or MPC), and it was decided to manually adjust the controller coefficients.

Furthermore, the process started with some reasonable guesses for the value of Kp, Ki, and Kd and then tried to find an acceptable combination step by step. The procedure for tuning the roll and yaw axis is as follows (Figure 7.1). Since the platform is symmetrical on the roll and pitch axis, it shows the same behavior on the pitch axis as it shows on the roll axis.

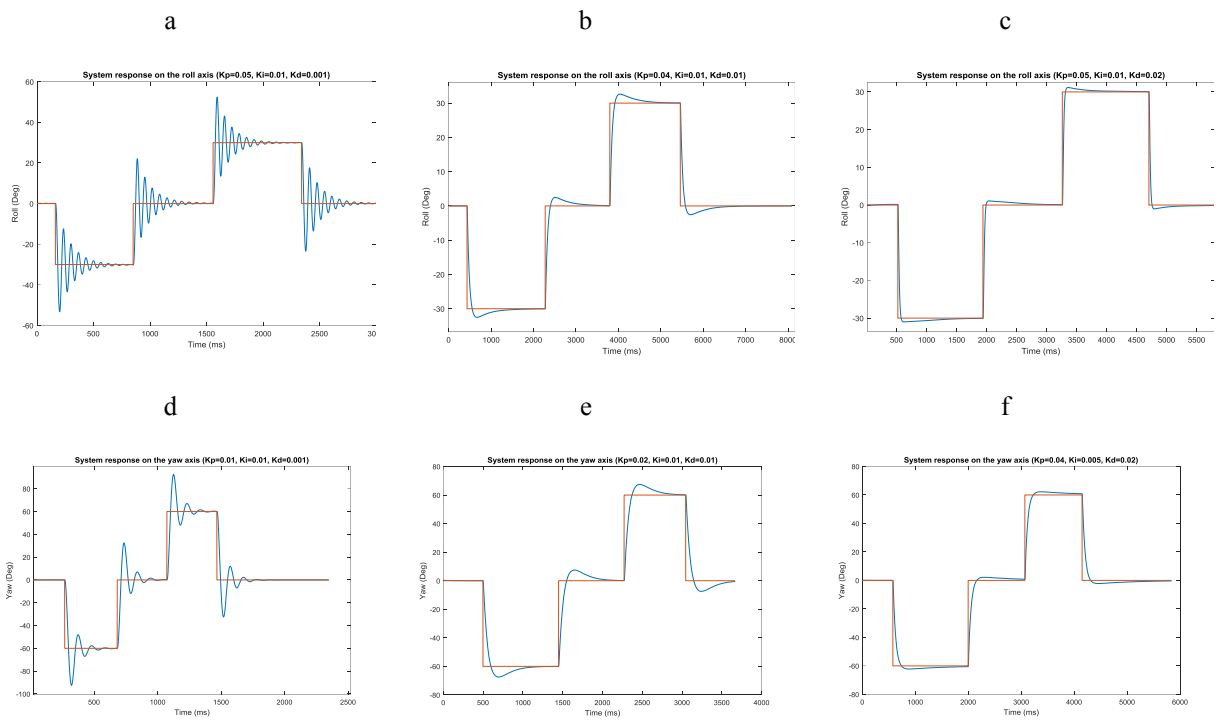


Figure 7.1: The System behavior and responses.

After following the same procedure for altitude and navigation controllers with the same PID structures as the others, it can navigate to a specific location. For example, the navigation strategy for the system is to change the platform heading to align with the target waypoint direction. When it gets alight (heading error < 5 degrees), it starts to move forward by changing the pitch angle. At the same time, it maintains the altitude, direction, and roll angle. Roll angle is always zero during the navigation. The roll and pitch controller diagram during the navigation is presented the Figure 7.2 and Figure 7.3. The waypoint navigation direction controller diagram is also depicted in Figure 7.4.

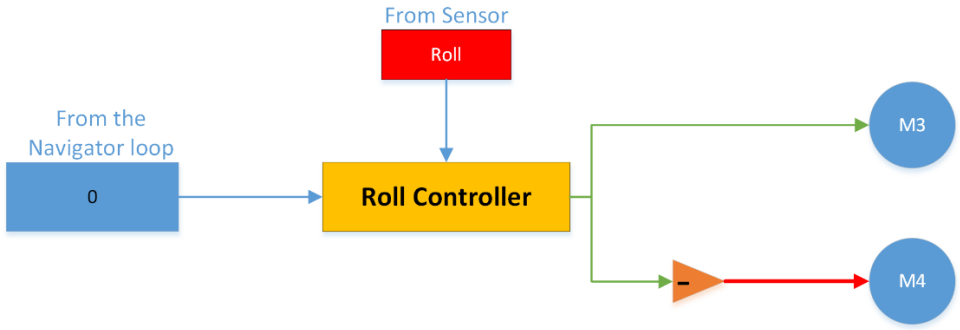


Figure 7.2: Roll Controller during navigation

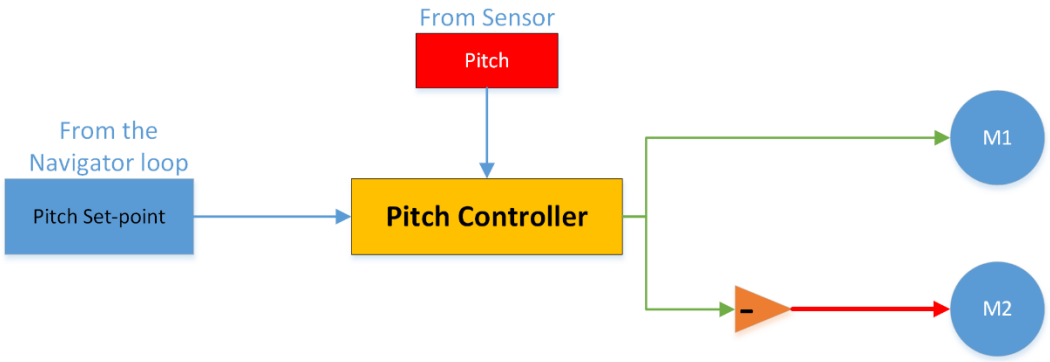


Figure 7.3: Pitch controller during navigation

The pitch set-point is a 15-degree constant while moving toward the target and is set to zero when it arrives at the waypoint to have enough time to change the direction to prevent overshoots. In other words, if it does not set it to zero, it gets an enormous overshot after touching each waypoint.

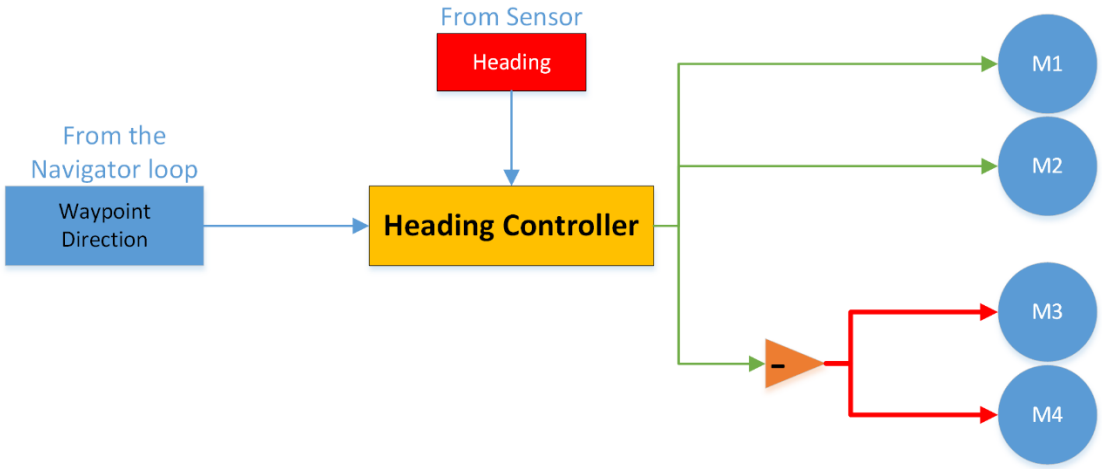


Figure 7.4: Navigation loop direction controller diagram

The waypoint direction is calculated in the navigation loop. The navigation loop maintains a list of the path waypoints. Each waypoint has an index number. When it starts or restarts the mission, the index is zero; when the platform distance to the waypoints is closer than a threshold value (distance < 10 meters), it changes the heading set-point to the next waypoint by adding one to the index number. The course between two points in the ECEF system (Lat, Lon, and the altitude) can be calculated using Eq.(1) :

$$X = atan2(\sin \Delta\lambda \times \cos \varphi 2, \cos \varphi 1 \times \sin \varphi 2 - \sin \varphi 1 \times \cos \varphi 2 \times \cos \Delta\lambda) \quad (1)$$

where :

φ = latitude and λ = longitude of each point

φ1, λ1 is the platform position, φ2, λ2 is the waypoint (Δλ is the difference in longitude)

The above formula can be converted to a C++ code in the following form:

```
// lat1, lon1 from GPS, and lat2, lon2 are waypoint coordination
y = sin(lon2-lon1) * cos(lat2);
x = cos(lat1)* sin(lat2) - sin(lat1)*cos(lat2)*cos(lon2-lon1);
θ = atan2(y, x);
wpBearing = (θ*180/Math.PI + 360) % 360; // course in degrees
```

Furthermore, to calculate the distance to the waypoint, the navigation loop uses Eq.(4).

$$a = sin^2(\Delta\varphi/2) + \cos \varphi 1 \times \cos \varphi 2 \times sin^2(\Delta\lambda/2) \quad (2)$$

$$c = 2\ atan2(\sqrt{a}, \sqrt{(1-a)}) \quad (3)$$

$$distance = R \times c \quad (4)$$

where R is the earth's radius, for simplifying the equation, the navigation loop uses the mean radius of earth that is 6378137 meters. In the same manner, the above formula can be converted to a C++ code as follows:

```
R = 6378137 ; // metres
f1 = lat1 * PI/180; // f1, f2, dF, dL has to be in radians
f2 = lat2 * PI/180;
dF = (lat2-lat1) * PI/180;
dL = (lon2-lon1) * PI/180;
a = sin(dF/2) * sin(dF/2) + cos(f1) * Mcos(f2) * sin(dL/2) * sin(dL/2);
c = 2 * atan2(sqrt(a), sqrt(1-a));
d = R * c; // in metres
```

7.3 Image prosseing method

The next step is to test the image processing system's functionality. The approach for detecting the fire location is to use the raw image from the simulated thermal camera and calculate

the centroid of high-temperature areas (the white pixels). Figure 7.5 shows the image that the flight controller sees and processes.

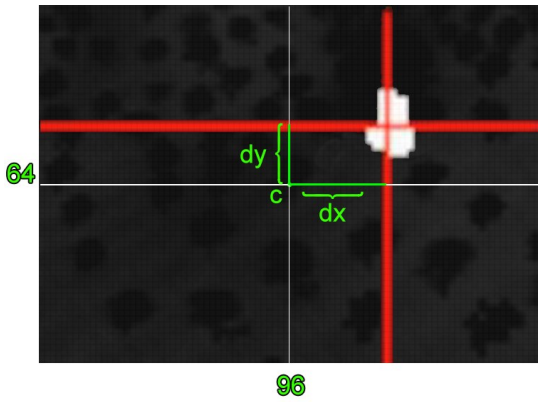


Figure 7.5 High-temperature area's centroid

Suppose the centroid position is off-center, based on its deviation from the image center, dy, dx (depicted in Figure 7.5), camera mounting angle, and the other camera parameters such as field of view. It estimates the fire location relative to the drone's absolute location that it gets from GPS and reports it to the GCS.

Given the slight roll angle during a normal flight, the enormous size of the fire, the low field of view of the camera, and assuming the terrain field flat, the equation can be simplified. This simplification has the least effect on the accuracy since, in a real application, even a couple of meter accuracy is still acceptable for fire detection and monitoring purposes. The equations as follows:

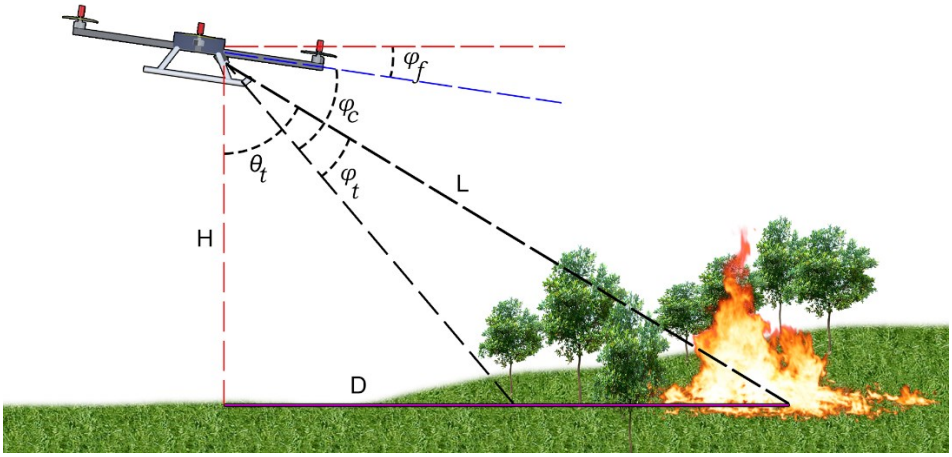


Figure 7.6: Side view geometry.

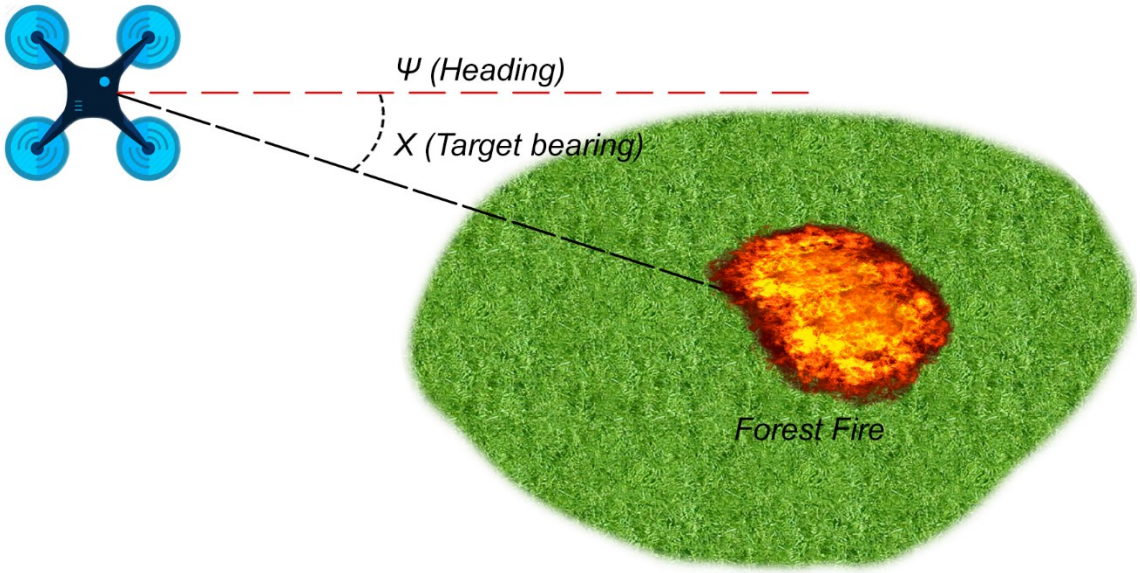


Figure 7.7: Top view geometry

For a 96H x 64V pixel image and 75°H x 60°V field of view, we have:

$$X_c = \frac{1}{n}\sum_{i=1}^{n} x_i \tag{5}$$

$$Y_c = \frac{1}{n}\sum_{i=1}^{n} y_i \tag{6}$$

$$dx = \frac{96}{2} - X_c \tag{7}$$

$$dy = \frac{64}{2} - Y_c \tag{8}$$

$$X_t = dx * \frac{75}{96} \tag{9}$$

$$\omega_t = \omega_{GPS} + X_t \tag{10}$$

$$\varphi_t = dy * \frac{60}{64} \tag{11}$$

$$\theta_t = 90 - (\varphi_f + \varphi_c) + \varphi_t \tag{12}$$

$$D_t = H_{GPS} * \tan\theta_t \tag{13}$$

And now that we have the X_t and θ_t we can use this equation[43] to calculate the fire location:

$$Lat_t = \arcsin(Sin(Lat_{gps}) * Cos\left(\frac{D_t}{R}\right) + Cos(Lat_{GPS}) * Sin\left(\frac{D_t}{R}\right) * Cos\omega_t) \tag{14}$$

$$Lon_t = Lon_{GPS} + Atan2(Sin\omega_t * \sin\left(\frac{D_t}{R}\right) * Cos(Lat_{GPS}), Cos\left(\frac{D_t}{R}\right) - Sin(Lat_{GPS}) * Sin(Lat_t)) \tag{15}$$

$x_i, y_i = x, y\ location\ of\ each\ pixel\ representing\ high\ temptature\ zone.$

$n = number\ of\ total\ pixels\ in\ the\ high\ temptature\ zone.$

$\omega_t = abslute\ target\ bearing$

$D_t =\ Estimated\ target\ ground\ distance\ from\ the\ drone$ *(Meter)*

$Lat_{gps} = The\ drone\ Latitude\ from\ GPS$

$Lon_{gps} = The\ drone\ Longitude\ from\ GPS$

$R = Earth\ radius =\ 6371 * 10^3\ meter$

7.4 HIL simulation result

For testing the HIL simulation, a specific maneuver was designed. After detecting the fire location, it performs a loiter maneuver around the fire while keeping the camera (the platform heading) pointed toward the fire. It calculates the target direction using the fire position from Eq. (14) and Eq. (15) and the current platform position from GPS (simulated) and sends it to the YAW controller as the set point. On top of that, we calculate the distance from the fire location and control it simultaneously. The result has been shown in Figure 7.8.

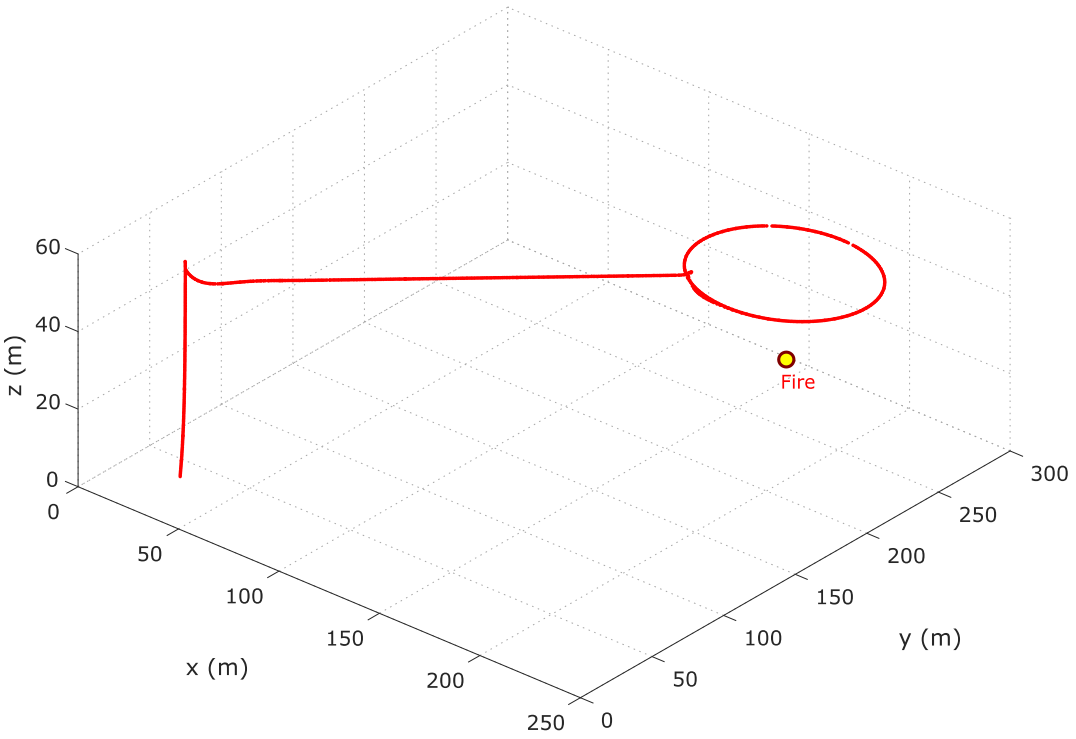


Figure 7.8 The flight path of the mission

The results suggest that the system has a high consistency where the guidance and navigation algorithms and the simulated image processing system perform to the satisfactory. We perform several other tests with different fire locations and calculated the error. The result is shown in Several other tests with different fire locations are carried out where the reported location error is calculated. The results are shown in Table 7.1. Since the terrain elevation is not provided to the system (the area is assumed to be a flat surface), thus only the x and y are calculated, and z is considered 0.

Table 7.1 Fire location estimation accuracy

Attempt	Actual location(x,y,z)	Detected location (x,y,z)	Error
#1	195, 219, 21	197.3, 231.4, 0	12.6m
#2	150, 100, 10	153.1, 107.3, 0	7.9m
#3	100, 100, 10	103.9, 106.1, 0	7.2m
#4	250, 250, 0	252.8, 254.5, 0	5.3m

7.5 Outdoor tests

The last confirmation for any simulation system is to test the result in a real-world situation, using real hardware in an uncontrolled environment. There are several requirements that need to be provided to the system to make it fly-worthy and tastable in an actual world situation.

7.5.1 Hardware setup

A quadcopter has a simple airframe structure. Fortunately, It is easy to construct using simple materials and skills. The frame is constructed using aluminum square tubes and fitted to for the motors and the autopilot installments screws. The platform needs four motors too. Brushless motors are the most efficient and common choices. For the designed weight (1200 grams), a motor of 400 watts has been chosen. The motor is depicted in Figure 7.9, and the specification is mentioned in Table 7.2.

Figure 7.9: SunnySky X2212

Table 7.2: The motor specification

SunnySky X2212 KV980	
Stator Diameter	22mm
Stator Thickness	12mm
Motor Kv (RPM/Volt)	980
No-load current	0.7A/10V
Max Continuous Current	25A/30s
Max Continuous Power	412W
Weight	56g

Each brushless motor needs a driver too. Brushless motor drivers are called electronic speed controllers (ESC). ESC is an electronic board capable of monitoring each motor's behaviors and driving them to meet the user's rpm command (in this case, the commands come from the autopilot). The ESC is depicted in Figure 7.10.

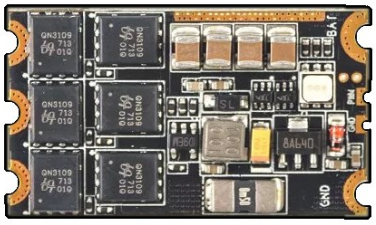


Figure 7.10: ESC

Furthermore, To test the thermal vision, a thermal sensor is needed. For this stage, a cheap 8x8 pixel (64 pixels) module has been chosen. The module is AMG8833 from Panasonic [44]. It transfers the image using an I2C interface.

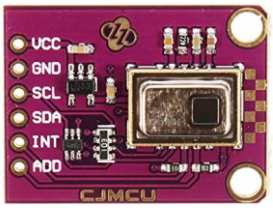


Figure 7.11: AMG8833 Module

Alongside the thermal sensor, a video camera has been chosen to record the video to inspect the sensor result. The installation of the two devices is shown in Figure 7.12.

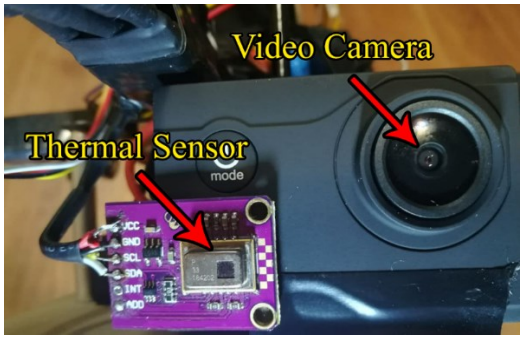


Figure 7.12: Thermal sensor and video camera installation

For safety reasons, an RC Radio controller (Figure 7.13) is needed too. The Radio controller allows the user to intervene and control the platform manually. For example, the user can kill the motor or bring the drone back if it was heading far away or a dangerous area.

Figure 7.13: Radio and the receiver

Finally, to connect the autopilot to the GCS, a pair of RF modems are needed (Figure 7.14). One modem connects directly to the GCS computer using a USB port, and the other connects to the autopilot using a TTL UART port.

Figure 7.14: RF modem

In the end, the system structure is depicted in Figure 7.15, and also an image of the assembled drone is shown in Figure 7.16.

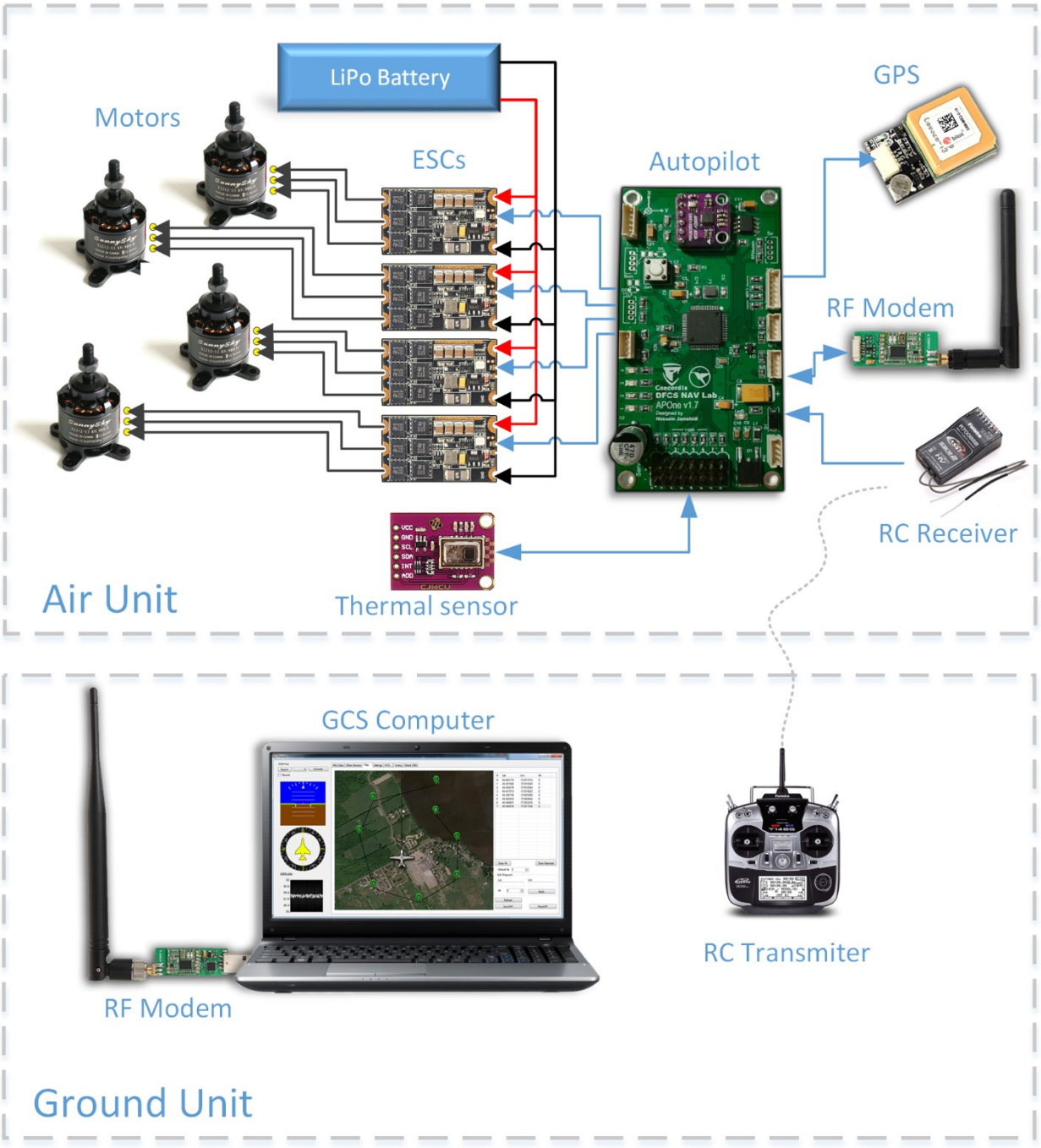


Figure 7.15: System architecture, air, and ground

Figure 7.16: Assembled drone

7.5.2 Flight tests

The flight test was involved with some autonomous flight to test the navigation system then flight to test the thermal sensor capability and detecting the fire location. The flight test was done in a safe and secure area. An outdoor grill was used as the heat source (Figure 7.17).

Figure 7.17: Heat source

After some primitive tests, the autonomous flight was started with a manual takeoff. After that, the autonomous flight started the mission; the mission included two waypoints in different altitudes (Figure 7.18). The results show that the navigation loop was performing fine, but the navigation loop needs some tuning to make the perfect navigation. In addition, there was some altitude bias in the sensors, but the overall performance was satisfying.

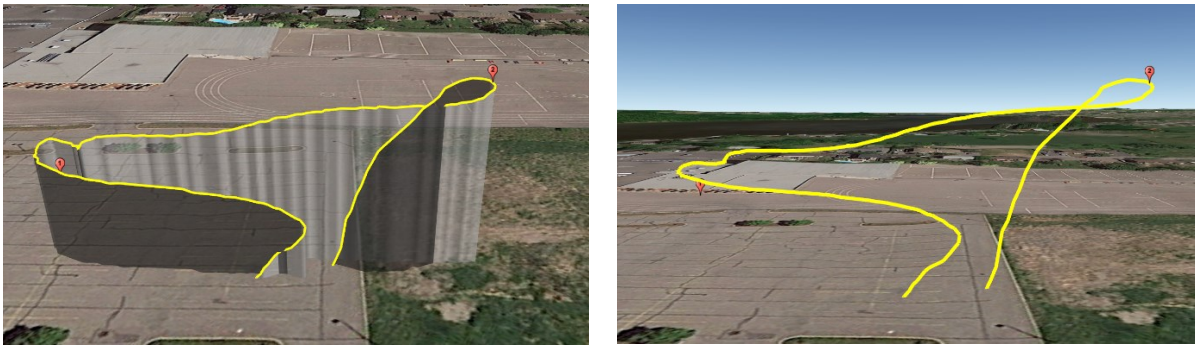


Figure 7.18: The mission Flightpath

The next test was the thermal sensor inspection. The test was involved with the drone flying above the grill (Figure 7.19) and check the heat signature and estimate the fire location (Figure 7.20). Because the sensor was a cheap sensor and didn't have some high performance that is needed for a flying robot, the outdoor performance was weak, but it was able to detect the grill from a considered distance. The farthest distance that it could detect the grill heat signature was 21 meters. Given the small size of the grill heat signature, it shows a fine performance even with the existing cheap sensor.

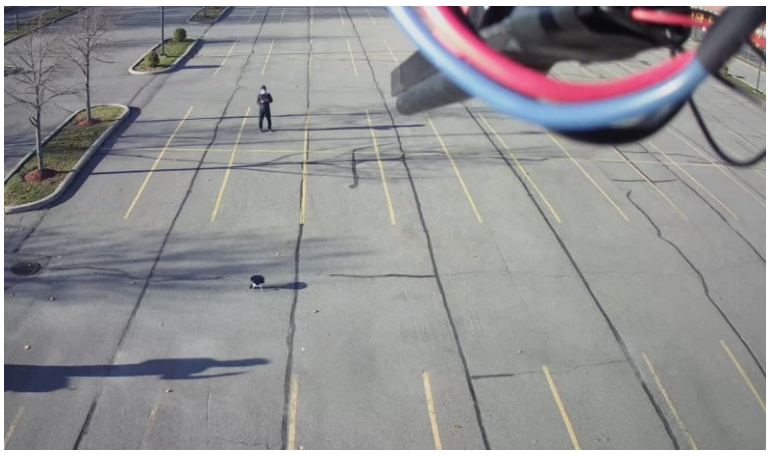


Figure 7.19: Outdoor flight onboard video

Figure 7.20: The fire detection result

Table 7.3: The fire detection and the location estimation result

The heat source actual coordination	Lat = 45.459151, Lon = -73.919035
The drone position	Lat = 45.459063, Lon = -73.918916
The fire estimated coordination	Lat = 45.459175, Lon = -73.919067
The drone distance from the grill	21 meter (13 meters on the ground, altitude = 17 meters)
Estimation error	~ 3.9 meters

The above result was the best system performance and was achieved after several rounds of testing. Even though the systems need more real-world testing to get the perfect result, it satisfies the testing goals to check the system consistency; all the subsystems work fine and are functional. A further test was obstructed by the COVID-19 pandemic and was not safe to perform since it needed more than one person to be performed.

7.6 Summary

The goal of this section was to test and confirm the system's performance. First, a control system is designed to test the platform stabilization; then, it expanded to altitude hold control and navigation controller. After getting the navigation system done, a simple fire detection algorithm

is designed to test the simulated thermal camera image and check its behavior. The simulation benchmark results show acceptable performance. Furthermore, the flying drone was assembled and perform the autonomous flight. The results show that the drone is flying worthy, and the navigation system works fine. After that, a fire location estimation test was performed. An artificial heat source was used to simulate the fire. Even though the system was under-tested (due to the COVID-19 pandemic limitation), it showed a high level of consistency and yet estimated the fire location with an acceptable error.

Chapter 8

Conclusion and future work

In the following chapter, the outcomes of this research are summarised. Furthermore, the intended future works are recommended. Finally, I hope these results can be used in other projects and eventually help to preserve our precious forests.

8.1 Results and conclusion

This research program started with the idea of making a drone to detect the fire before it gets outrageous. A fast response needs a fast notification. It started with designing the autopilot. The autopilot was design base on one of the author's previous designs. The new autopilot is called APOne v1.7. It was powered by an STM32F722 microcontroller and utilize several sensors that an autopilot needed. Having a functional autopilot needs to implement some sophisticated algorithms such as state estimation, control, and waypoint navigation. Like every other computer program, those codes need to be tested. Also, the embedded implementation needs to be tested; thus, a test platform needs to be implemented; Such a test system that considers the real hardware in the testing loop is called hardware-in-the-loop or HIL. Because of this research's nature and intention and the final goal that is to design a solution to detect fire using the onboard thermal cameras, a new HIL system is needed to be designed since such a complete system is not part of any free or even affordable systems. It started with designing a simple setup to just implement the 3D system base, 6DOF dynamics, and the interfaces. The 3D system is implemented using the Unity engine. Unity engine is a well-known game engine. The 6DOF dynamic and eventually the flight dynamic model is power by using the built-in Nvidia Physix engine. The interfacing is the most sophisticated part of this research. Data latency a key role in this system. If you are planning

to design such a system, make sure you did all the research and have all the abilities. When all the above task is done properly, it is time to simulate the forest and the fire. Since the actual fire behaviors are too complicated to be calculated in a real-time system, thus the fire is a pre-rendered animation with the help of the Unity particle system. To confirm the system consintency, a flight scenario is needed. The scenario was to flight to a pre-programmed direction and estimate the fire location (if any). The test was successfully full, and the final results and the estimated location error are presented for each round of testing. Despite the limitations (due to simplifications), the fire location estimation shows a rational error and confirms the system's consistency. The final test was to test the real platform. The platform is a quadcopter similar the drone used in the HIL simulation. Some outdoor tests confirm the system's general performance. For making a perfect platform, some more tests are needed, but unfortunately, the tests were not feasible due to the COVID-19 pandemic.

8.2 Contribution

Overall, in the course of this research, a simulation environment for a fire detection drone is designed. The fire and the thermal camera simulation can help other researchers to test their search algorithms and benchmark their efficiency. The equation that is developed to transfer the object's coordination in the image to the real-world location using the drone position and attitude is an academic achievement that can be used on other research projects. This equation shows a good performance while running it on the MCU and doesn’t need any range detector to full fill the formula. Furthermore, a durable autopilot for almost all robotic projects is design and manufacturing; it can be modified or expanded to be a solid starting point for other mechatronic systems.

8.3 Future work

Following the research challenges and according to the difficatties and the limitation were faced during this research, there are several steps needed to be done to make the final product more operational.

- Use a better thermal camera. The current thermal sensor that was used has a limited performance during daylight and losses accuracy under sunlight. A Lepton v3.5 LWIR is a good yet affordable candidate [45].
- To add a camera Gimbal. The gimbal helps to have broader and steadier vision feedback.
- Add a laser range finder to detect the exact fire location. This sensor needs to be coupled with the gimbal system to detect the distance with the object in the camera image center. Keep in mind that this type of sensor is expensive (and heavy); it can significantly increase the overall cost.
- Use a more accurate method to sense the platform altitude. During the outdoor tests, a significant altitude bias made the auto take-off and landing impossible. To tacle this issue, a laser range finder can be used to help with take-off and landing, but if the goal is to make it solid, it is better to use an RTK GPS. Even though the RTK GPS is expensive and needs a base station, it allows the system to get a position feedback accuracy of about a centimeter for vertical and horizontal positioning [46][47].
- More importantly, practice more outdoor tests when the COVID-19 pandemic is over!
- Finally, when the platform is ready, it is time to design a command center network to connect all the units to a central command and monitoring system. Figure 8.1 shows the command center UI mockup.

Figure 8.1: A mockup of the command and monitoring center software.

Appendix A

Codes

A.1 PID controller C++ Class source code

```
/**
 * Implementation of PID Controller
 * @author Hossein Jamshidi
 * Contact: hosseinjamshidien@gmail.com
 *
 */
class PID{

    double iMax;
    double iMin;
    double dMax;
    double dMin;
    double dState;
    double iState;
    double lastFeedback;
    double outputMax;
    double outputMin;
    boolean iTermLimit = false;
    boolean dTermLimit = false;
    boolean outputLimit = false;
    double outTemp = 0;
    double deltaFeedback = 0;
    double pGain;
    double iGain;
    double dGain;
public:
    double pValue = 0;
    double dValue = 0;
    double iValue = 0;
    double err = 0;
    PID(){
        iState = 0;
        pGain = 0;
        iGain = 0;
        dGain = 0;
        iTermLimit = false;
```

```
    last_micros = 0;
}
void reset(){
    iState = 0;
}

/**
 * Sets the controler gains.
 *
 * @param _pGain the controller P gain.
 * @param _iGain the controller I gain.
 * @param _dGain the controller D gain.
 */
void setK(double _pGain, double _iGain, double _dGain){
    pGain = _pGain;
    iGain = _iGain;
    dGain = _dGain;
}
/**
 * Limits the controller I term in order to prevent high overshoots.
 *
 * @param _iMin the controller, I term minimum output.
 * @param _iMax the controller I term maximum output.
 */

void set_ITermLimits(double _iMin, double _iMax){
    iMax = _iMax;
    iMin = _iMin;
    iTermLimit = true;
}

/**
 * Limits the controller D term in order to limit noise effects.
 *
 * @param _dMin the controller D term minimum output.
 * @param _dMax the controller D term maximum output.
 */
void set_dTermLimits(double _dMin, double _dMax){
    dMax = _dMax;
    dMin = _dMin;
    dTermLimit = true;
}

/**
 * Limits the controller output to limit it to the actuator constraints.
 *
 * @param _oMin the controller minimum output.
 * @param _oMax the controller maximum output.
 */
void set_outputLimits(double _oMin, double _oMax){
    outputMax = _oMax;
    outputMin = _oMin;
    outputLimit = true;
}

/**
 * Limits the controller output to limit it to the actuator constraints.
```

```
     *
     * @param feedback the current reading of the sensor.
     * @param desire the setpoint of the controller.
     * @param dt elapsed time between calculation in uS.
     * @return the controller output.
     */
    double updatePID(double feedback, double desire, double dt){
        if (dt<1){ dt = 1; } // to deal with the timer roll over.
        dt /= 1000000.0F; // convert uS to to Sec

        err = desire - feedback;
        iState += err * dt;
        pValue = pGain * err; // calculate the P term.
        iValue = iGain * iState; // calculate the I term.

        if (iTermLimit){// limit the I term.
            if (iValue > iMax) {
                iValue = iMax;
            }
            if (iValue < iMin) {
                iValue = iMin;
            }
        }

        deltaFeedback = lastFeedback - feedback;
        dValue = dGain * (deltaFeedback) / dt; // calculate the D term.

        if (dTermLimit){// limit the D term.
            if (dValue > dMax) {
                dValue = dMax;
            }
            if (dValue < dMin) {
                dValue = dMin;
            }
        }

        lastFeedback = feedback;
        outTemp = pValue + iValue + dValue;

        if (outputLimit){
            if (outTemp > outputMax) outTemp = outputMax;
            if (outTemp < outputMin) outTemp = outputMin;

        }
        return outTemp;
    }

};
```

References

[1] IPCC, "IPCC SR: Climate Change and Land," An IPCC Spec. Rep. Clim. Chang. Desertif. L. Degrad. Sustain. L. Manag. food Secur. Greenh. gas fluxes Terr. Ecosyst., 2019.

[2] NASA, "NASA: Climate Change and Global Warming," NASA, 2021. https://climate.nasa.gov/ (accessed Jun. 29, 2021).

[3] NASA, "Earth's Atmospheric Layers | NASA," NASA, 2017. https://www.nasa.gov/mission_pages/sunearth/science/atmosphere-layers2.html (accessed Jun. 26, 2021).

[4] N. Geographic, "atmosphere | National Geographic Society," Geographic, National, 2021. https://www.nationalgeographic.org/encyclopedia/atmosphere/ (accessed Jun. 26, 2021).

[5] G. of Canada, "CLIMATE CHANGE AND FIRE MANAGEMENT," jourenal Artic., 2020, Accessed: Jun. 29, 2021. [Online]. Available: https://www.nrcan.gc.ca/our-natural-resources/forests-forestry/wildland-fires-insects-disturban/climate-change-fire/13155.

[6] J. Mataix-Solera, C. Guerrero, F. García-Orenes, G. M. Bárcenas, and M. P. Torres, "Forest fire effects on soil microbiology," in Fire Effects on Soils and Restoration Strategies, 2009.

[7] H. Aaltonen et al., "Forest fires in Canadian permafrost region: the combined effects of fire and permafrost dynamics on soil organic matter quality," Biogeochemistry, 2019, doi: 10.1007/s10533-019-00560-x.

[8] A. Chenebert, T. P. Breckon, and A. Gaszczak, "A non-temporal texture driven approach to real-time fire detection," in Proceedings - International Conference on Image Processing, ICIP, 2011, pp. 1741–1744, doi: 10.1109/ICIP.2011.6115796.

[9] B. U. Töreyin, Y. Dedeoğlu, and A. E. Çetin, "Flame detection in video using hidden Markov models," in Proceedings - International Conference on Image Processing, ICIP, 2005, vol. 2, pp. 1230–1233, doi: 10.1109/ICIP.2005.1530284.

[10] NASA, "Atmosphere Discipline Team Imager Products," NASA, 2021. https://atmosphere-imager.gsfc.nasa.gov/ (accessed Jun. 26, 2021).

[11] ArduPilotDevTeam, "ArduPilot Open Source Autopilot," Ardupilot.org. 2018.

[12] PX4, "Pixhawk," PX4, 2021. https://pixhawk.org.

[13] STMicroelectronics, "STM32F103CB - Mainstream Performance line, Arm Cortex-M3 MCU with 128 Kbytes of Flash memory, 72 MHz CPU, motor control, USB and CAN - STMicroelectronics," STMicroelectronics, 2015. https://www.st.com/en/microcontrollers-microprocessors/stm32f103cb.html#documentation (accessed Jun. 29, 2021).

[14] STMicroelectronics, "STM32F7 - ARM Cortex-M7 Microcontrollers - STMicroelectronics," 2021. https://www.st.com/en/microcontrollers-microprocessors/stm32f7-series.html (accessed May 22, 2021).

[15] STMicroelectronics, "STM32F722xx Datasheet," 2020. Accessed: May 25, 2021. [Online]. Available: www.st.com.

[16] Invensense, “MPU-9250 | TDK,” 2015. https://invensense.tdk.com/products/motion-tracking/9-axis/mpu-9250/ (accessed May 23, 2021).

[17] Motorola, “SPI Block Guide V04.01,” 2004. Accessed: Jun. 22, 2021. [Online]. Available: www.freescale.com.

[18] Bosch Sensortec, “BMP280 Technical Specification,” 2020. Accessed: May 25, 2021. [Online]. Available: www.bosch-sensortec.com.

[19] u-blox, “SAM-M8Q Easy-to-use u-blox M8 GNSS antenna module Data sheet Document information Title SAM-M8Q Subtitle Easy-to-use u-blox M8 GNSS antenna module Document type Data sheet Document number This document applies to the following products: Product name Type number ROM/FLASH version PCN reference,” 2020. Accessed: May 31, 2021. [Online]. Available: www.u-blox.com.

[20] U.S. government, “GPS.gov: New Civil Signals,” U.S. government, 2020. https://www.gps.gov/systems/gps/modernization/civilsignals/ (accessed Jun. 29, 2021).

[21] Wikipedia, “Serial port - Wikipedia,” Wikipedia, 2021. https://en.wikipedia.org/wiki/Serial_port (accessed Jun. 29, 2021).

[22] Faa, “FAA-H-8083-15B, Instrument Flying Handbook,” 2020. Accessed: May 31, 2021. [Online]. Available: http://www.faa.gov/regulations_policies/.

[23] Microsoft, “System.Drawing.Drawing2D Namespace | Microsoft Docs,” Microsoft, 2020. https://docs.microsoft.com/en-us/dotnet/api/system.drawing.drawing2d?view=net-5.0 (accessed Jun. 29, 2021).

[24] Google, “Google Maps Platform,” Google, 2020. https://cloud.google.com/maps-platform/terms (accessed May 31, 2021).

[25] Justin Stoltzfus, “What is Multithreading?,” Techopedia, 2021. https://www.techopedia.com/definition/24297/multithreading-computer-architecture (accessed Jun. 29, 2021).

[26] M. Rouse, L. George, and M. Chuck, “What is UDP (User Datagram Protocol,” TechTarget, 2020. https://searchnetworking.techtarget.com/definition/UDP-User-Datagram-Protocol (accessed Jun. 29, 2021).

[27] C. Kleijn, “Introduction to Hardware-in-the-Loop Simulation,” 2020.

[28] Wikipedia, “Unity (game engine),” 2020. https://en.wikipedia.org/wiki/Unity_(game_engine) (accessed May 23, 2021).

[29] R. Mahony, V. Kumar, and P. Corke, “Multirotor aerial vehicles: Modeling, estimation, and control of quadrotor,” IEEE Robot. Autom. Mag., vol. 19, no. 3, pp. 20–32, 2012, doi: 10.1109/MRA.2012.2206474.

[30] Unity, “Rigidbody overview,” Unity, 2021. https://docs.unity3d.com/Manual/RigidbodiesOverview.html (accessed May 21, 2021).

[31] J. Fingas, “NVIDIA’s physics simulation engine,” 2018. https://www.engadget.com/2018-12-03-nvidia-physx-open-source.html (accessed May 21, 2021).

[32] Unity, “Continuous collision detection (CCD),” Unity, 2021. https://docs.unity3d.com/Manual/ContinuousCollisionDetection.html (accessed May 21, 2021).

[33] J. Zhu, “Conversion Of Earth-Centered Earth-Fixed Coordinates To Geodetic Coordinates,” IEEE Trans. Aerosp. Electron. Syst., 1994, doi: 10.1109/7.303772.

[34] Unity, “Introduction to Particle Systems - Unity Learn,” Unity, 2021. https://learn.unity.com/tutorial/introduction-to-particle-systems (accessed May 23, 2021).

[35] H. Sundell and P. Tsigas, “Fast and lock-free concurrent priority queues for multi-thread systems,” in Proceedings - International Parallel and Distributed Processing Symposium, IPDPS 2003, 2003, p. 11, doi: 10.1109/IPDPS.2003.1213189.

[36] W. Stallings, “The advanced encryption standard,” Cryptologia, 2002, doi: 10.1080/0161-110291890876.

[37] Wikipedia, “Firmware - Wikipedia,” Wikipedia, 2021. https://en.wikipedia.org/wiki/Firmware (accessed Jun. 29, 2021).

[38] Pinzaru, What is firmware? 2013.

[39] R. De Andrade, K. N. Hodel, J. F. Justo, A. M. Laganá, M. M. Santos, and Z. Gu, “Analytical and Experimental Performance Evaluations of CAN-FD Bus,” IEEE Access, vol. 6, pp. 21287–21295, Apr. 2018, doi: 10.1109/ACCESS.2018.2826522.

[40] Wikipedia, “I^2C,” Wikipedia, 2021. https://en.wikipedia.org/wiki/I^2C (accessed Jun. 29, 2021).

[41] K. H. Ang, G. Chong, and Y. Li, “PID control system analysis, design, and technology,” IEEE Trans. Control Syst. Technol., vol. 13, no. 4, pp. 559–576, Jul. 2005, doi: 10.1109/TCST.2005.847331.

[42] S. Bennett, “A Brief History of Automatic Control,” IEEE Control Syst., vol. 16, no. 3, pp. 17–25, 1996, doi: 10.1109/37.506394.

[43] C. Veness, “Calculate Distance and Bearing between Two Latitude/Longitude Points Using Haversine Formula in JavaScript,” MIT Open Source. pp. 1–39, 2002, Accessed: May 10, 2021. [Online]. Available: http://www.movable-type.co.uk/scripts/latlong.html.

[44] Panasonic Industry Europe GmbH, “AMG8833,” Panasonic, 2020. https://industry.panasonic.eu/components/sensors/industrial-sensors/grid-eye/amg88xx-high-performance-type/amg8833-amg8833 (accessed Jun. 29, 2021).

[45] Teledyne FLIR, “Lepton LWIR Micro Thermal Camera Module,” FLIR, 2020. https://www.flir.ca/products/lepton/ (accessed Jun. 29, 2021).

[46] NovAtel, “Real-Time Kinematic (RTK) ,” NovAtel, 2021. https://novatel.com/an-introduction-to-gnss/chapter-5-resolving-errors/real-time-kinematic-rtk (accessed Jun. 29, 2021).

[47] G. C. Weiffenbach, “Tropospheric and Ionospheric Propagation Effects on Satellite Radio-Doppler Geodesy,” in Electromagnetic Distance Measurement, University of Toronto Press, 2019, pp. 339–352.