NATIONAL UNIVERSITY OF HO CHI MINH CITY
HO CHI MINH CITY UNIVERSITY OF TECHNOLOGY
FALCUTY OF TRANSPORTATION ENGINEERING
DEPARMENT OF AEROSPACE ENGINEERING

-------o0o-------

FINAL REPORT FOR GRADUATION PROJECT

IMPLEMENTATION A FLIGHT COMMAND FOR A QUADROTOR

Student: Nguyễn Anh Quang
Student ID: V1002583
Supervisor: Professor Emmanuel GROLLEAU

Poitier, Spring 2015

Lời cam kết - Commitment

Tôi cam kết:

- Đây là luận văn tốt nghiệp do tôi thực hiện.
- Các số liệu, kết quả nêu trong luận văn là trung thực và chưa từng được ai công bố trong bất kỳ công trình nào khác.
- Các đoạn trích dẫn và số liệu kết quả sử dụng để so sánh trong luận văn này đều được dẫn nguồn và có độ chính xác cao nhất trong phạm vi hiểu biết của tôi.

I hereby commit that:

1. This report is my graduation report and is done by me and only me.
2. Every word, every number and conclusion shown in this report is correct and done by me with the supervisor of Professor Emmanuel GROLLEAU at LIAS, ISAE-ENSMA, France. This data is the results of my Projet de Fin d'Étude (PFE) at LIAS in 2015 and has not been published in any publication.
3. Every reference and paraphrasing used in this report will be cited fully and clearly with the name of the original author, the documents where they have been found and other related information with the highest accuracy in my knowledge.
4. If the result in this report is found as lying, invalid or copied the work of others, I will take full responsibility and willing to take any punishment necessary.

Author,

NGUYỄN Anh Quang

Intentionally left blank

Lời cảm ơn - Acknowledgements

In first place, I would like to give my truly appreciation to professor Emmanuel GROLLEAU at **L**aboratoire d'**I**nformatique et d'**A**utomatique pour les **S**ystèmes (LIAS), ISAE-ENSMA, France for all of his supports and especially for giving me the opportunity to do this project.

I also want to say thank you to all of the professors and lecturers at Department of Aerospace Engineering, Ho Chi Minh City University of Technology for giving me priceless knowledge and for passing on me the passion with this major in the last few years.

For my friends, whether classmates, roommates or colleagues, thank you for the time we have passed together. There is no word in this world could describe truly and fully the things we have shared together. Without you, it is hard to imagine how I could have the chance to finish this project.

Lastly, but most importantly, I want to give the best words with all my heart to my family and my special one. You give me a shoulder when I was down and raise my up again, keep me strong and take me back to my way when I was lost.

Poitiers, 2015

Nguyễn Anh Quang

Intentionally left blank

Mục lục - Table of Contents

Tóm tắt luận văn

Ngày nay, cùng với sự phát triển của các ngành khoa học – kĩ thuật, máy bay không người lái (unmanned controlled vehicle – UAV) đang được nghiên cứu và hoàn thiện bởi nhiều nhóm và trường đại học khác nhau trên toàn thế giới. Trong số đó, các máy bay 4 chong chóng (quadrotor hay quadricopter) là một trong những chủng loại được quan tâm và đề cập đến trong nhiều công trình nghiên cứu khác nhau.

Đề tài của luận văn này, "Hiện thực hoá một lệnh điều khiển cho máy bay 4 chong chóng", là một sự tiếp nối và hoàn thiện 2 đề tài đã được thực hiện trước đây của BÙI Nhã Đạt và Ana Benito CASTILO. Sự khác biệt của đề tài so với hai đề tài đã có và các loại máy bay bốn chong chóng hiện hành, nằm ở các yếu tố sau:

- Bo mạch điều khiển được sử dụng là **FlyMaple**, một bo mạch mới, có nhiều ưu điểm so với các mạch điều khiển thế hệ cũ hơn. **ArduPilot** là nền tảng được chọn để xây dựng và phát triển chương trình lái tự động cho máy bay bốn chong chóng của đề tài này.
- Máy bay được điều khiển thông qua một trạm mặt đất (Ground Control Station) bằng giao tiếp không dây chứ không thông qua sóng vô tuyến thông thường. Giao thức được sử dụng là **MAVLink**.
- Trong quá trình tạo lập chương trình tự động điều khiển bằng ArduPilot, thêm một phần mở rộng đã được thêm vào đoạn mã gốc nhằm thực hiện việc đa dạng hóa các dạng thuật toán điều khiển có thể được sử dụng. Qua đó giúp cho việc so sánh và đánh giá khả năng điều khiển của các thuật toán điều khiển khác nhau.

Từ khóa: FlyMaple, MAVLink, Quadricopter, ArduPilot, ArduCopter, control algorithm

Intentionally left blank

Abstract

Nowadays, along with the development of aerospace engineering in general, Unmanned Aerial Vehicles has gained many achievements. Among the UAVs, quadricopter is one of the most interested types and has been studied and developed by many universities and groups around the world.

This project, ***"Implementation a flight command for a quadricopter",*** is the continuous of the two previous projects called ***"Flight Control and Navigation of a Quadcopter"*** of Ana CASTILLO BENITO and ***"Embedded System for Quadricopter"*** of BÙI Nhã Đạt. The differences of this project comparing with the two above as well as the other related quadricopter projects are

- This project uses **FlyMaple** as the control board. The platform used to create the firmware for the quadricopter is **ArduPilot**, one of the most popular framework for autopilot firmware.
- The UAV is controlled and followed by a ground control station. MAVLink is the communication protocol embedded in ArduPilot and will be used in this project.
- From the original code of ArduPilot, a new module will be created to help developers in the future integrating new control algorithms and test them with the simulator as well as the real quadricopter.
- This project will also complete the work of the two projects above and fly the complete model for the first time using new control algorithms.

Từ khóa: FlyMaple, MAVLink, Quadricopter, ArduPilot, ArduCopter, control algorithm

Intentionally left blank

Danh mục bảng biểu - List of table

Danh mục hình ảnh – List of figures

Các chữ viết tắt - Abbreviations

UAV	Unmanned Aerial Vehicle
UA	Unmanned Aircraft
FAA	Federal Aviation Administration
DOF	Degree of Freedom
GPS	Global Positioning System
RC	Radio Control
Wi-Fi	Wireless Fidelity
ESC	Electric Speed Controller
DC	Direct Current
PWM	Pulse Width Modulation
Li-Po	Lithium Polymer
APM	Arduino Mega
OSI	Open Systems Interconnection
I/O	Inputs/Outputs
GCS	Ground Control Station
UDP	User Datagram Protocol
TCP	Transmission Control Protocol
SITL	Software in the Loop
HIL	Hardware in the Loop
ID	Identification
MAV	Micro Air Vehicle
ack	acknowledgement
PID	Proportional Integral Derivative
AHRS	Attitude and Heading Reference System
IB	Integral Backstepping
wp	waypoint
GNSS	Global Navigation Satellite System
HUD	Head-up display
elf	executable and linkable format

Chú giải ký hiệu - Nomenclature

Symbols

F_{lift}	Total lift generated by the rotors	N
F_x	Horizontal lift	N
F_y	Vertical lift	N
F_g	Gravity force	N
X	Position in X axis from the origin	m
Y	Position in Y axis from the origin	m
Z	Position in Z axis from the origin	m
P	Proportional gains of the PID controller	-
I	Proportional gains of the PID controller	-
D	Proportional gains of the PID controller	-
U_1	Global thrust	N
U_2	Roll Torque	Nm
U_3	Pitch Torque	Nm
U_4	Yaw Torque	Nm
t	Time since the starting of the control board	s
dt	Time since the last time the command is called	s

Greek Symbols

Ω	Angular speed	rad/s
ϕ	Roll	radian
θ	Pitch	radian
ψ	Yaw	radian
$\dot{\phi}$	Roll change rate	radian/s
$\dot{\theta}$	Pitch change rate	radian/s
$\dot{\psi}$	Yaw change rate	radian/s

Subscripts

d	desired value

Chương 1

An introduction to Unmanned Aerial Vehicle and Autopilot

This chapter gives an overview about Unmanned Aerial Vehicle (UAV) as well as the autopilot system.

1.1 Unmanned Aerial Vehicle

According to a definition of the United States Army, Unmanned Aerial Vehicle (UAV), or Unmanned Aircraft (UA) is *"an aircraft that does not carry a human operator and is capable of flight with or without human remote control"* [**1**]. With many advantages comparing to other kinds of aircraft such as compatibility, mobility, safety for pilots…many countries and companies around the world have tried to develop and do researches related to the UAV.

There are different references about the beginning of the Unmanned Aerial Vehicles. Some references believe that the first Unmanned Aerial Vehicle was used on August 22, 1849 when the Austrians used balloons to attack the city of Venice [**2**]. Meanwhile, according to J.F.Keane and S.S.Carr, the using of radio signal to control aircraft was studied for the first time in 1911, just eight years after the first manned flight of Wright brothers [**3**]. The development of UAV,

like the development of aviation and many other technologies, is close to the requirements of the wars. During World War I, World War II and many other wars after that, UAVs have been used as drones to train the pilots, balloons to surveillance, etc.

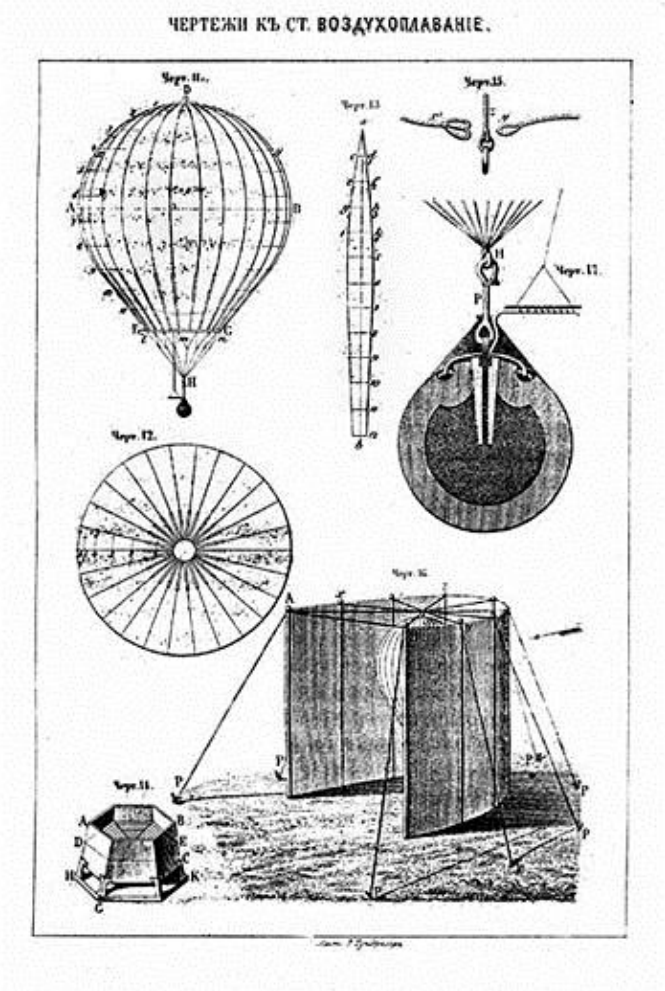


Figure 1-1: Bombing by Balloon, 1848
(**Source : Prof. Jurij Drushnin, Moscow, Russia)**

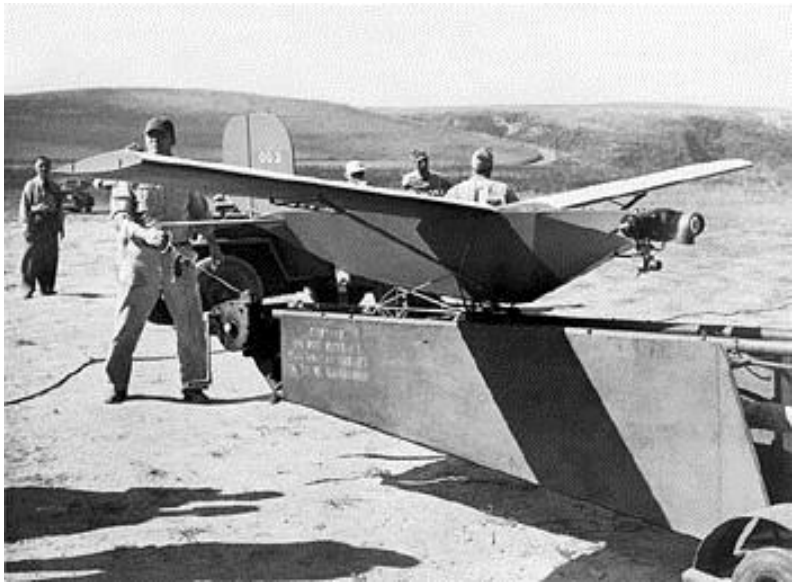


Figure 1-2: OQ-3, Righter O-15-3, El Paso, Texas, 1941(Source: **Righter Family Archives) [**4**]**

Figure 1-3: Reginald Denny, OQ-3 launch (1947) **[**4**]**

In 21st century, noticing the benefits the UAV, many countries and companies have studied this kind of aircraft. At this time, UAV can be found in many sizes, shapes and are suitable for various types of mission (surveillance, mapping, patrolling, scientific, weathering, etc.). Some applications of these aerial vehicles can be seen in **figure 1-4**.

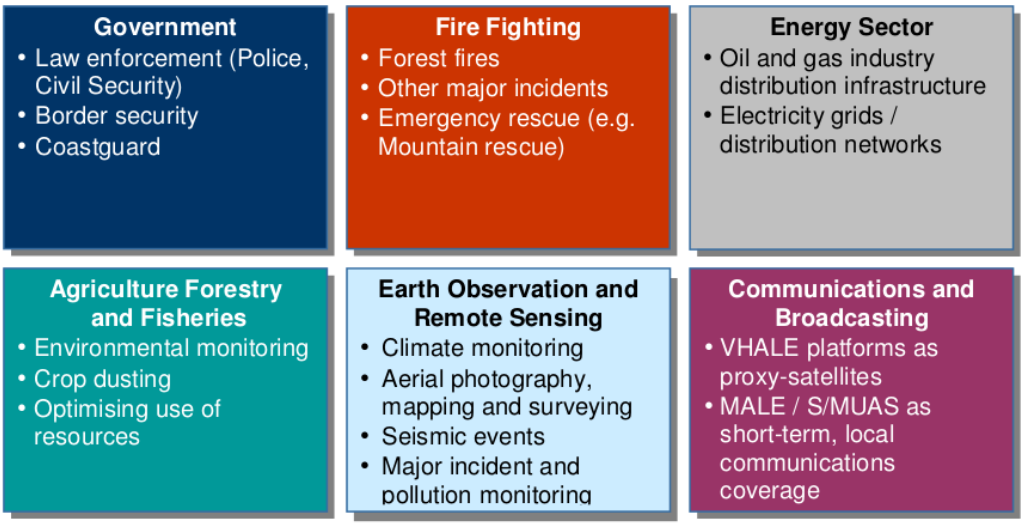


Figure 1-4: Applications of UAV [5]

1.2 Autopilot system

The development of UAV is relative to the development of autopilot. From the beginning using the wind to fly the balloons to later using radio signal to have a direct control of the drones from a distance, it can be concluded that the creation of autopilot system has brought UAV to a higher level. With autopilot systems, UAV can operate without the integration of human, the real unmanned status likes its name.

Unlike radio control, which is easy to implement and apply, an autopilot system requires many complicated supporting components. However, autopilot has a major advantage that other controlling methods cannot have. With the autopilot system, the UAV can operate out of sight and can stay in air for a very long duration.

The history of autopilot systems can be traced back to 1911 with the Sperry family. **E.A.Sperry** is an American inventor, who is famous as the co-author of the gyrocompass, one of the most important components for UAV and aircraft in general nowadays [**6**]. Gyrocompass is a type of non-magnetic compass, it is not affected by the magnetic field of the earth and have a very special behavior while the gyroscope is rotating. As can be seen from **figure 1-5**, there is a fixed axis that will not change while changing the direction of the leg. This behavior

will create an angle between two axes. By measuring this difference, the angle that the base axis has changed will be noticed. In 1914, **Lawrence Sperry** demonstrated a simple autopilot system operating the rudder and elevators in a straight level flight by flying an aircraft with hands away from the controls [**7**].

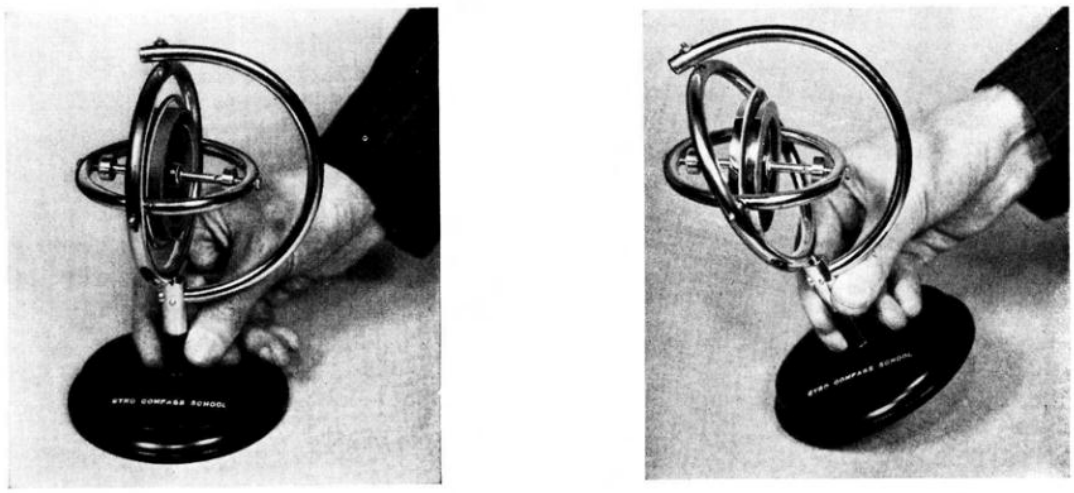


Figure 1-5: Gyrocompass behavior while the gyroscope is rotating

Later, **Elmer Sperry Jr**, a son of **Lawrence Sperry**, continued to develop the autopilot system. In 1930, he demonstrated an autopilot system that can *"keep an aircraft on true heading and altitude for three hours"* [**8**].

Figure 1-6: Controlled by Sperry gyroscopes, a pilotless flight in 1930 [9**]**

At this time, with the development of technology and science, autopilot system is much more precise, can handle multiple tasks at a time and of course much more complicated. According to the **Advanced Avionics Handbook of the Federal Aviation Administration** (FAA), *"an autopilot can be capable of*

many very time intensive tasks, helping the pilot focus on the overall status of the aircraft and flight. Good use of an autopilot helps automate the process of guiding and controlling the aircraft. Autopilots can automate tasks, such as maintaining an altitude, climbing or descending to an assigned altitude, turning to and maintaining an assigned heading…" [**10**] . Depending on the controlled systems, an autopilot might be as simple as some sensors and a controller or as complicated as a system including thousands of components. In general, the more components, the more data the system could gain at the same time, the more precise the decision based on this data.

Chương 2

An introduction to quadricopter and the components of the quadricopter in this project

This chapter includes the following parts:

- *A quick introduction about quadricopter*
- *Quadricopter movement in relation with rotors rotation*
- *The objectives of this project*
- *The components and modules of the developed quadricopter*

2.1 Quadricopter and its control theory

2.1.1 Quadricopter

Quadricopter, or quadrotor, is one of the major researched and developed UAVs due to its simplicity and versatility. As it is called, this type of drone has four equally-spaced rotors, which can be divided into two groups basing on their rotating direction: clockwise and counter clockwise. Comparing with other types of flight vehicle, quadricopter has some benefits as well as drawbacks. In this report, quadricopter, quadrotor and quad are used with the same meaning.

Quadricopter is a multi-copter vehicle; therefore, it can vertically take off as well as landing. Moreover, since its rotors are symmetric and rotate clockwise, counter-clockwise in pairs, the gyroscopic[1] effect is reduced and even is canceled in hover. Besides, there are many other advantages of this type of UAV such as flexibility, small-space suitability, etc.

On the other hand, there are some minus for this type of UAV. Like any other flight vehicles, quadricopter has 6 **d**egrees **o**f **f**reedom (6 DOFs), three translations and three rotations. Meanwhile, it is controlled by only four-fixed rotors, which means it is an underactuated and dynamic unstable system. In order to control the system, these four rotors have independent angular speeds, the calculation for these values will be therefore complicated, in particular for an unstable systems. As a result, the controllability, especially finding an optimized control algorithm for this type of UAV is still an interesting topic for developers in this field. This project is one of those, which is focused on creating a simple solution to integrate different control algorithms for a quadricopter into an existing firmware generator and then comparing the results to find out the pros and cons of each controller.

[1] Gyroscopic effect is the resistance to change in the direction of rotation axis. In other copters such as helicopter, this effect causes the gyroscopic precession, more detail about this can be found [**58**]

2.1.2 The quadricopter dynamics

Based on the relation between the movement direction and the positions of the rotors, there are two groups of quadricopters, each of them has separated dynamics model and has to be handled differently. **Figure 2-1** introduces the name as well as the orientation of these groups. Although the numbering can be different depending on the developers, the rotating direction and the type of propeller must be the same in most of the cases. This project will only focus on the plus frame quadricopter, the details of the other type can be found in [**11**] and [**12**].

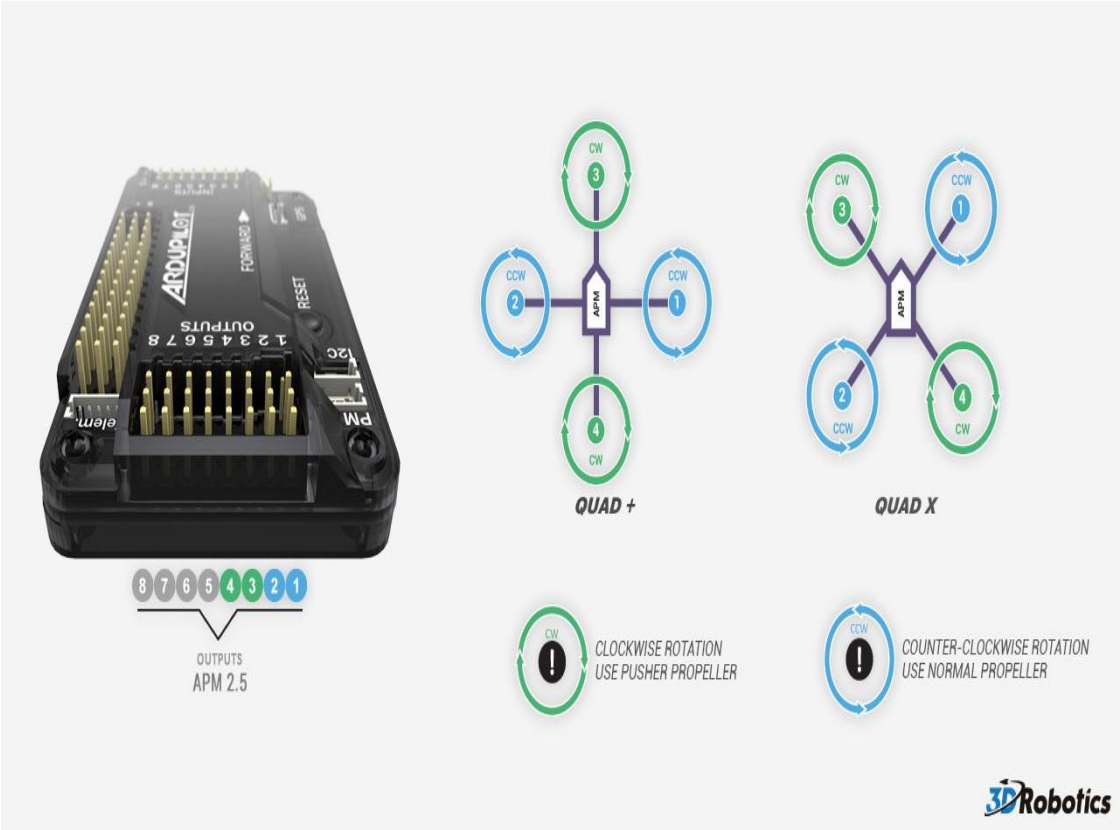


Figure 2-1: Two types of quadricopter and the number of each motor [13]

The movement of the quadricopter can be divided into two groups: altitude and attitude control. With the former, by increasing or reducing the rotating speed of each motor with the same amount, the lift force generated by each propeller will be changed equally, the system can level up or down respectively.

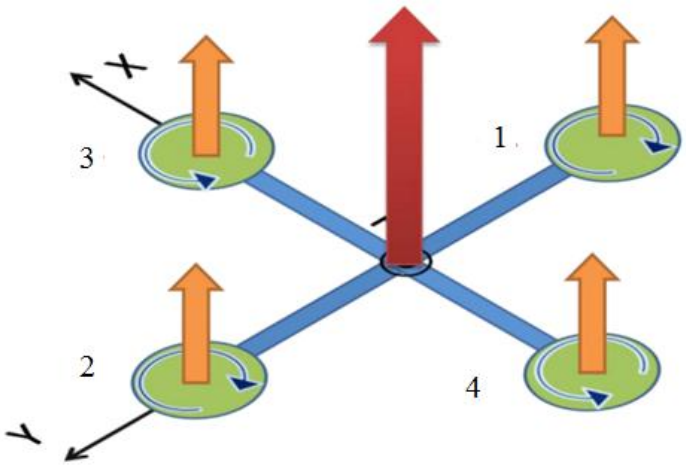


Figure 2-2: Altitude control [14]

When the adjustments of each motor are different, we can have the latter. Because of the unequal forces of each propeller, the system will change its attitude and its roll, pitch and yaw relative angles can be controlled. **Figure 2-3**, **figure 2-5** and **figure 2-6** present the basics of each mode.

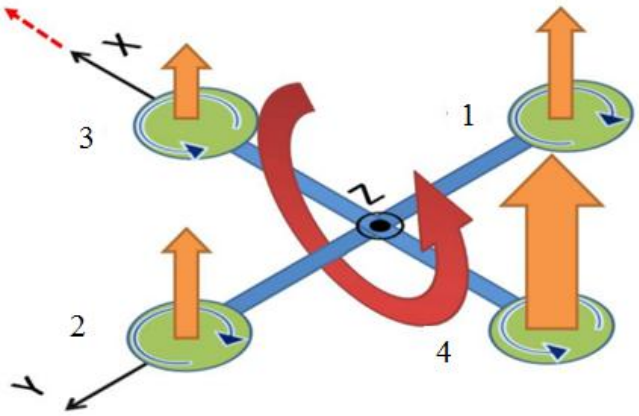


Figure 2-3: Pitch control [14]

In pitch control, motor one and two share the same speed, meanwhile, motor three is rotating a little slower (or faster) than its partner does. Consequently, the forward side of the quad will have less (or more) lift and the whole system will tilt forward (backward), creating a pitch angle. The total lift force generated by all four rotors can be now decomposed into two components. The vertical one will be in charge of controlling the altitude of the system; meanwhile, the horizontal will push the system to move forward.

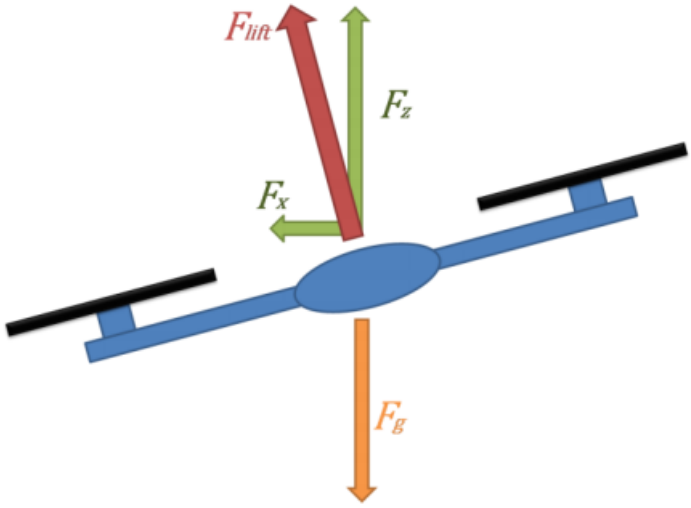


Figure 2-4: Lift force decomposed into a z-axis component and a horizontal component [14]

The same thing happens in case of roll control, however this time the differences occur between motor one and two.

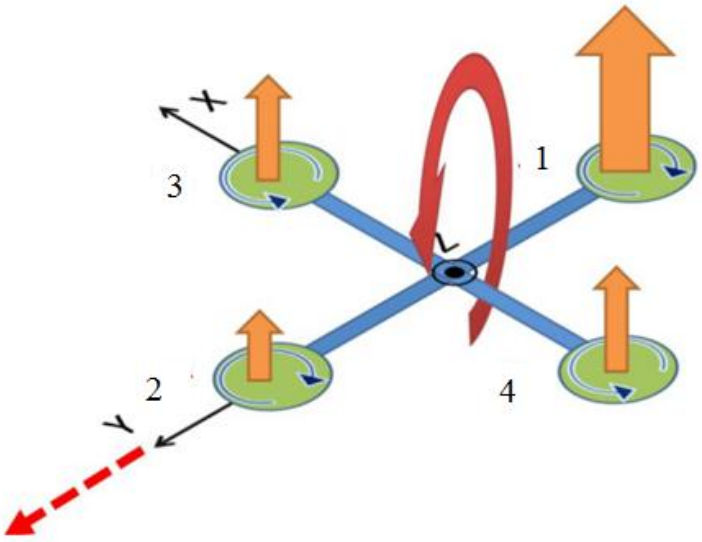


Figure 2-5: Roll control

With the yaw control, instead of making the different rotating speed in one pair of rotor, the difference is now between two pairs. Moreover, in order to change the yaw, lift is not the one taking into account but drag is. By generating the differences in rotating speed between two pairs, the system will have the forces as shown in **figure 2-6**. The higher the lift is, the higher the drag. As a

result, a rotate momentum will be created and make the quadricopter rotate around its vertical axis.

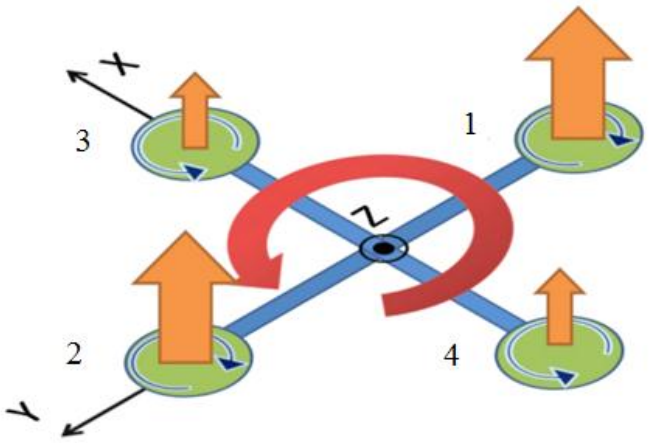


Figure 2-6: Yaw control [14]

More details and about these control as well as the dynamics equations of the plus frame quadricopter can be found in the previous work [**14**].

2.2 Components and modules of a quadricopter

2.2.1 General concepts and definitions

In general, an UAV, which can be operated with autopilot, includes the following modules:

- The embedded system

This can be considered as the heart of the drone. It commonly consists of a control board, the sensors, communication module and the GPS module. Depending on the use of the drone, these modules might be changed or there might be other modules. However, this system will be in charge of controlling and operating the drone, with or without the guide of human.

Autopilot is a very complicate system, which requires a lot of different data and calculation at a time. An accelerometer, a magnetometer, a gyroscope

and a barometer are the basics of the sensors system. **Table 2-1** gives an introduction about the data provided by these sensors.

Table 2-1: Basics sensors and their functions

Sensors	Data
Accelerometer	Proper acceleration: acceleration relative to freefall (g-force) and the attitude in x,y,z axis
Magnetometer	The strength of the magnetic field in different direction, generally used to detect magnetic north
Gyroscope	Angular velocity and angle
Barometer	Pressure, temperature and altitude

For a manually controlled UAV, GPS module is not required. However, in case of using autopilot, this is a must-have component since it will provide the control board the location of the system. Based on this location and the prepared flight plan, the autopilot will decide what to do next and generate the signals to control the drone.

Communication module is another important one. Even though the UAV with autopilot can operate on its own, there must be a way to communicate between the drone and its operators. This connection is not only for getting and receiving data but also for safety reasons. There are several ways to communicate with the drone. For example, **r**adio **c**ontrol (RC) is the most simple one. RC communication can be added into the system by attaching a RC receiver. In some cases, Wi-Fi or Bluetooth connection can be used to establish the communication.

Among the above components, control board is the most important. From getting the data from the sensors and GPS systems, handling the communication between components and systems to calculating the motor control signals and saving the required data, control board must take care of all of these functions. The strength of the autopilot depends mainly on this control board, for example, its memory, its computational ability, its communication protocol between components, etc.

However, putting these modules together is just a complete hardware of the embedded system. Since these components cannot operate on their own, there

must be a firmware taking the control, particularly the calculations. Firmware is *"a software program or set of instructions programmed on a hardware device and provides the necessary instructions for how the device communicates with the other computer hardware"* [**15**]. Generating a suitable firmware for the hardware is one of the most important works for any UAV project.

- The frame

Frame can be considered as the physical body of the UAV, keeping other modules at their place and supporting the whole system. In spite of their simple function, this module has a crucial effect to the controllability and stability of the drone. One reason is that the shape, the size of this frame is usually the shape and the size of the drone. Another reason is that the weight of the frame, in case of an UAV, will take at least a quarter of the total weight. More importantly, since the frame is the one putting every module together, the durability of the frame will be one of the most crucial limits for every calculation.

- The propulsion system

This system will be in charge of providing the required forces. For a quadricopter, this system includes four electric motors. Most of the case, these motors will be DC brushless motors. In order to control these motors, each motor has to be connected with an **E**lectric **S**peed **C**ontroller (ESC). **Figure 2-7** introduces a simple connection between ESC, controller board, rotor and battery. Among the three wires connecting ESC and control board, the yellow one is the signal wire, the red is the power wire and the black is the ground wire.

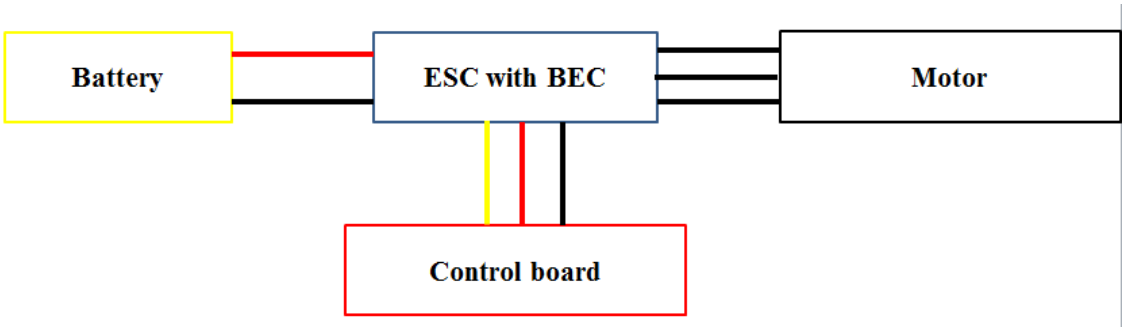


Figure 2-7: Connection between Battery, ESC, Control board and Motor

By sending the Pulse Width Modulation (PWM) signal to the ESC, control board can control the rotating speed of the rotor and therefore have the desired forces.

Another component of this system is the propeller. For quadricopter, there are two pairs of propeller have to be used. As can be seen in **figure 2-1**, the propellers attaching with rotor No.3 and No.4 must be push propeller, meanwhile, the others are normal propeller (in some documents they are called pull propeller). This is because of the differences in the rotating direction of each pair of motors. Using push-pull propeller properly will make sure that the forces generated by each rotor will be the same if they have the same rotating speed

Figure 2-8: Normal propeller (top) and push propeller (bottom)

- The energy system

Even though there are many energy sources can be used for an UAV, for a quadricopter, battery, particularly Lithium Polymer (Li-Po) battery is used in most of the cases. Even though there are many benefits of this type of battery comparing with the others, according to many references and manuals, Li-Po battery is considered as a danger and can cause fire in some case. Extra precaution must be taken while working with Li-Po batteries.

- The electric system

The electric system for quadrotor is simple comparing with other types of UAV. **Figure 2-9** presents a simple but detail method for wiring between components.

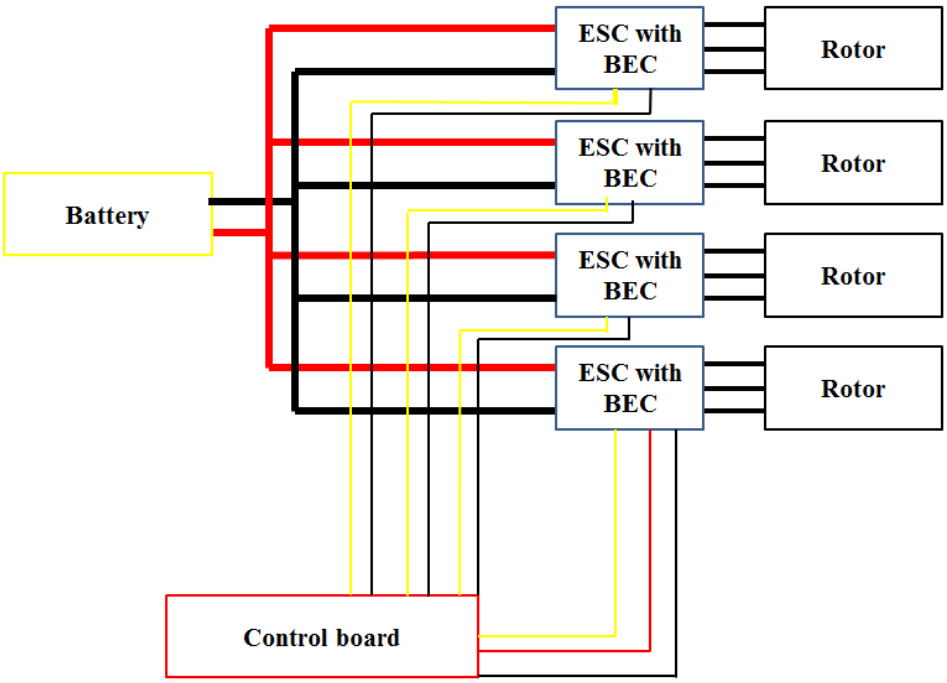


Figure 2-9: Quadricopter wiring

The most important note for the figure above is that **there is only one ESC connecting with the control board with all three wires**. The others use only two out of three, which are the ground and the signal wire. The control board will share its power with other shields such as GPS shield, external sensors, etc. Connecting more than one red wire from the ESC might damage these shields and the control board due to overload.

2.2.2 The components of the quadricopter in this project

2.2.2.1 Embedded system

- Hardware

In this project, **FlyMaple** is the control board. This is a 10DOFs controller with internal barometer, accelerometer, gyroscope and magnetometer. One of the strength of this control board is that it can be operated at 72MHz, which is

extremely fast comparing with other board. Other components of the hardware include a Wi-Fi shield and a GPS shield with external antenna to increase the strength of the GPS system. More detail about the hardware of the embedded system can be seen in [**16**].

- Firmware

Firmware of this project is created with **ArduPilot**, a popular and wide-used framework to generate firmware for UAV. The architecture of ArduPilot, its advantages, its drawbacks as well as the required modifications for a FlyMaple will be discussed in **chapter three**. The communication protocol between the quadricopter and the ground control station will be MAVLink. Details about this protocol can be found in **chapter four**.

2.2.2.2 The frame and other components

A full list of components using in this project as well as their characteristics can be seen in **Appendix A**.

Chương 3

An overview about ArduPilot

This part gives an overview about the structure of ArduPilot. It will show how its layered software architecture has allowed its portability and how it relies on the native drivers of the target boards. In order to understand the procedure of building a firmware using ArduPilot, the information related to cross-compiled and GNU build can be found in Appendix B.

3.1 Ardupilot

3.1.1 Overview

"ArduPilot and ArduPilot Mega (APM) is an open source autopilot system supporting multi-copters, traditional helicopters, fixed wing aircraft and rovers" [**17**]. As open-source software, it is developed and is contributed to by a large number of programmers around the world for many types of drone, hardware as well as different versions of firmware. It is used worldwide as a common software platform for programming and controlling the autopilot systems.

The name ArduPilot comes from the combination of Arduino, which is a very popular company providing the board (hardware) and the language to communicate with the hardware (software).

At this time, ArduPilot supports Fixed-wing plane (ArduPlane), multicopters, traditional helicopters (ArduCopter) and ground vehicles

(APMrover2). However, knowing the basics of this framework, users can create the autopilot for almost any unmanned vehicles. The full list of projects related to ArduPilot can be found in [**17**].

As mentioned, ArduPilot can be used in various hardware. Some specific control boards being supported by the development team of ArduPilot are Pixhawk, PX4FMU, FlyMaple, APM, etc. This project uses FlyMaple as the control board because it is available at the lab.

3.1.2 ArduPilot subdomains

ArduPilot includes more than seven hundred thousand code lines just for the basic codes [**18**], which can be divided into five different sub-components as in **Figure 3-1**.

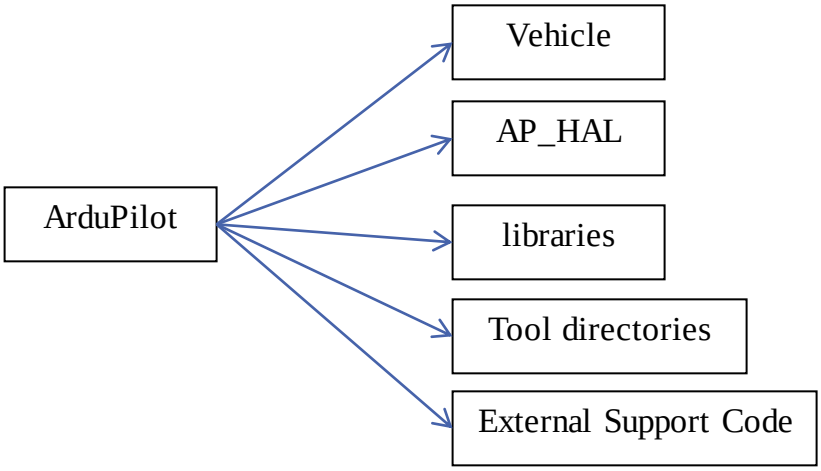


Figure 3-1: Structure of ArduPilot codebase

The first component is **"Vehicle"**, it contains any related code determining the unmanned autopilot systems, which are ArduPlane, ArduCopter and APM2rover, as mentioned above.

"AP_HAL" defines the code glue used to target specific main boards. Using Doxygen[1] to generate the documentation of ArduPilot, we can have **fig 3-2**.

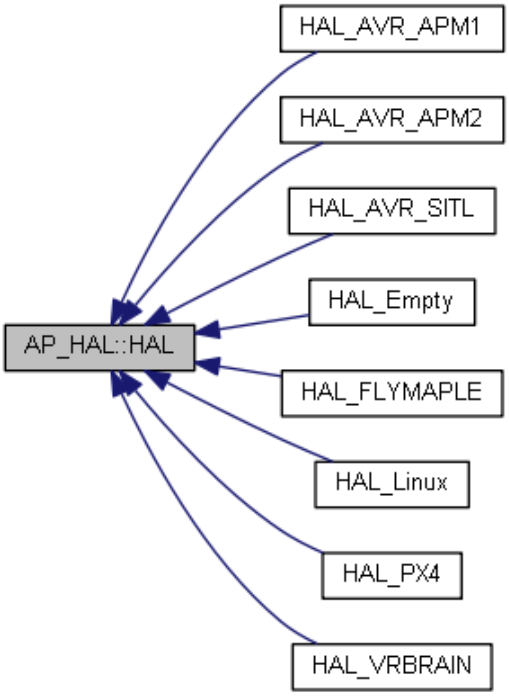


Figure 3-2: AP_HAL structure and the supported boards

From the figure above, it can be seen that ArduPilot supports various types of boards. Even if the board is not in the list, ArduPilot can be ported to it by specializing as "AP_HAL_EMPTY".

"Libraries" are the most important part of this framework, giving users the functions and codes to communicate with and control the board. For example, for a quadricopter, the libraries provide users the required tools to calculate the Pulse Width Modulation signals to control the rotor, to communicate with the controller, GPS system, sonar systems, etc.

[1] Doxygen is a tool for generating documentation from annotated C++ sources, but it also supports other popular programming languages such as C, Objective-C, C#, PHP, Java, Python, IDL (Corba, Microsoft, and UNO/OpenOffice flavors), Fortran, VHDL, Tcl, and to some extent D. More information about this tool can be found at http://www.stack.nl/~dimitri/doxygen/

"Tools directories" gives users the tools to do some system checking (sensors, GPS…). It also includes the tools used for simulation, testing new control algorithms. This simulation will be discussed later in this report. **"External Support Code"** supports code needed for new generation of main boards, such as Pixhawk.

3.1.3 Basic structure

ArduPilot is constructed as a multilayer software platform. In concept, a multi-layer structure can be described as in **fig 3-3**.

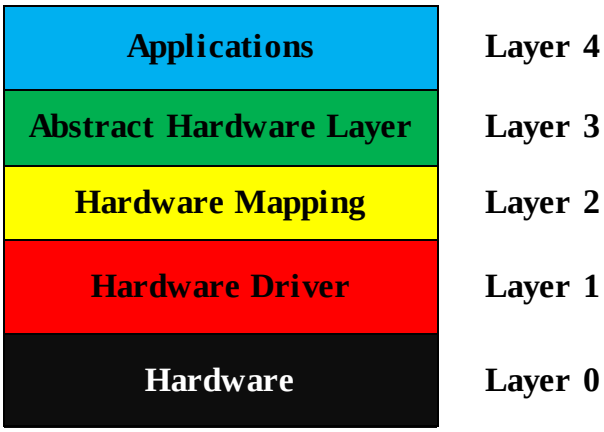


Figure 3-3: Multi-layers structure

Multi layered architectures are widely used in hardware independent application. One popular example is the **O**pen **S**ystems **I**nterconnection (OSI) model, which is a conceptual model describing the common idea for many communicating systems, the internet for example. The idea of this model is that the system can be built from different physical components; allow different hardware can communicate with each other regardless the underlying structure. This is essential for a network, like the internet, because it does not limit the availability of the hardware. That is one of the most important benefits making communication systems such as internet and mobile phone network has developed rapidly in a very short time.

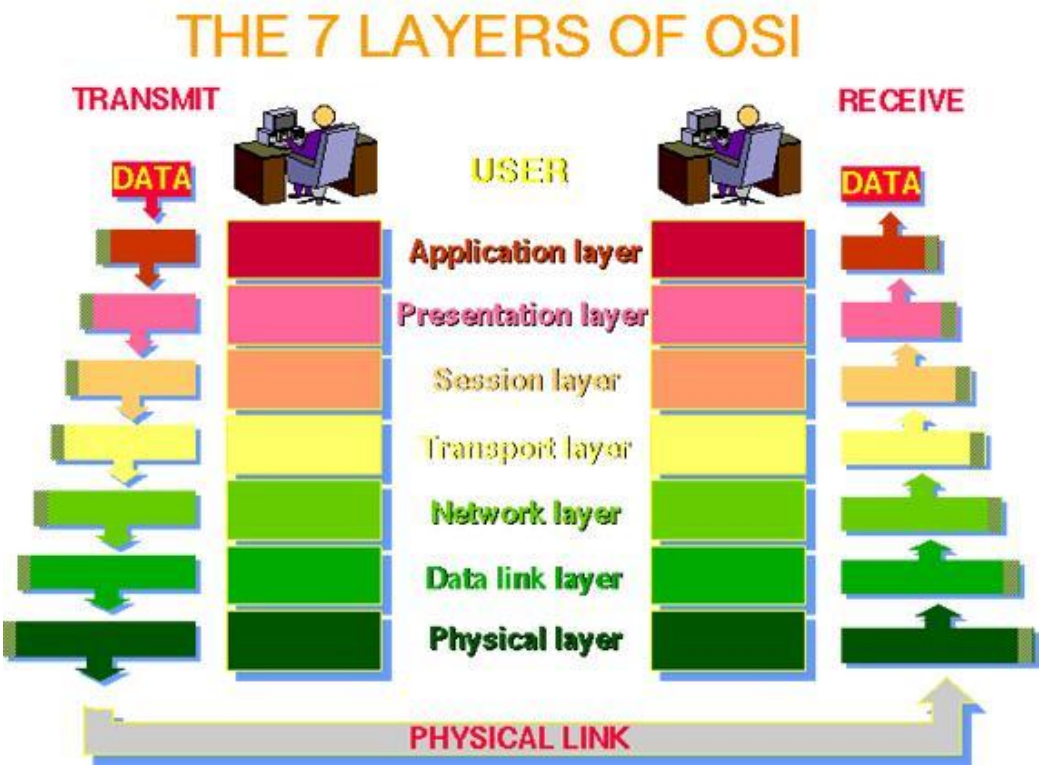


Figure 3-4: Multi layered structure of OSI model [19]

The principle of the multilayer structure is that the layer n uses the services provided by layer n-1 and then contributes its own services to the next n+1 layer. With the structure in the **figure 3-3**, **hardware** is **layer 0**, which is the physical layer. **Layer 1** is the **Driver** specific for the hardware in layer 0, using the facilities of the hardware and providing the next layer the required protocols to communicate with the hardware. The next layer, **Hardware Mapping** will use these protocols and then offer its own services to the **Abstract Hardware Layer**. The advantages of this architecture, as can be seen is that layer n is only depends on the services offered by layer n-1. In other words, **Application**, the highest layer, is independent from the **Hardware.** This will make the application become multi-targets, reduce the amount of works needed to make a compatible version of that application for other hardware. If we replace the hardware, which comes with its native drivers, only the hardware mapping layer has to be adapted for the application to work

Figure 3-5 presents the multi-layers structure of ArduPilot on a FlyMaple

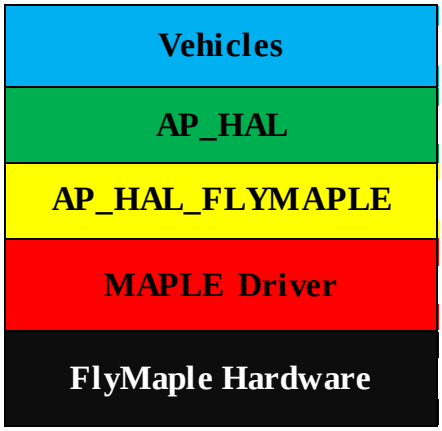


Figure 3-5: ArduPilot structure with FlyMaple hardware

The Vehicles layer includes the command for specific vehicle such as ArduPlane, ArduCopter. AP_HAL is the abstract layer, with the general functions in **figure 3-6**.

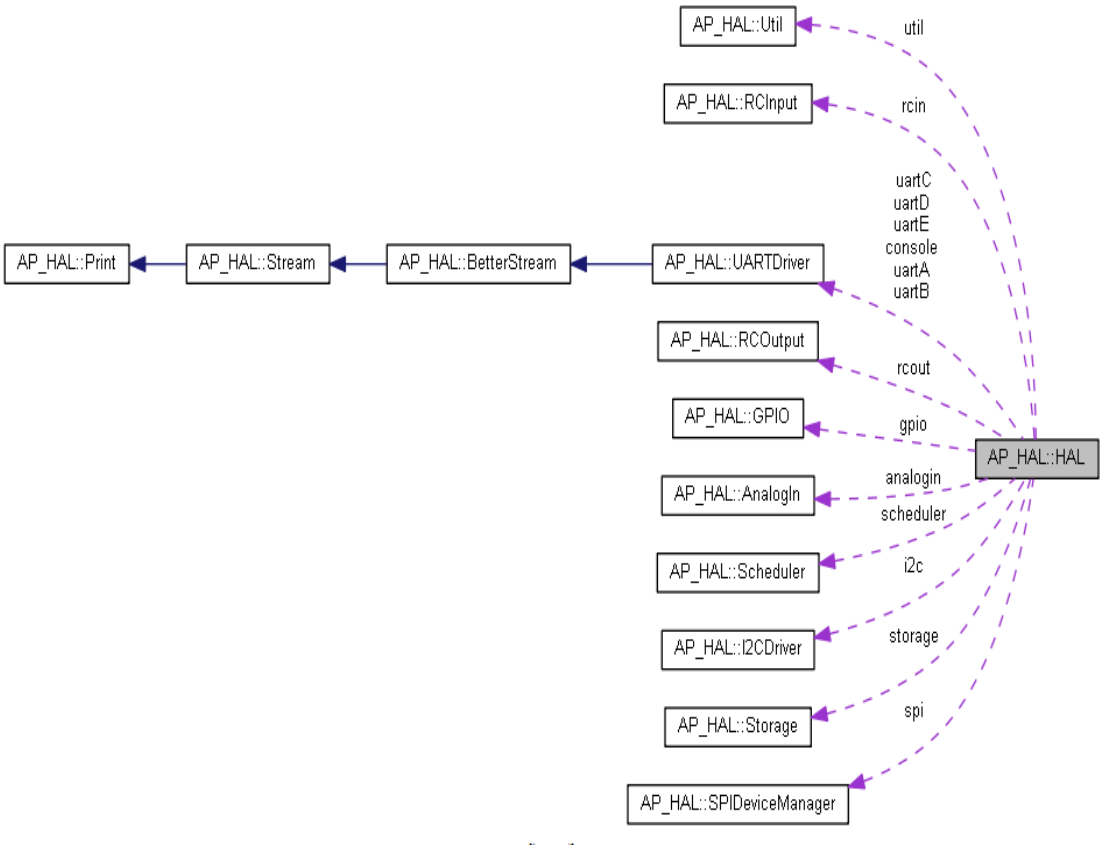


Figure 3-6: General functions in the abstract hardware layer

Programmers will program their autopilot functions based on this layer, which include nothing but the virtual commands. **AP_HAL_FLYMAPLE** connects the abstract systems in this layer to the correct place in FlyMaple board, implementing the virtual commands. As can be seen from **figure 3-6**, AP_HAL

defines serial ports from **uartA** to **uartE** and the **console** as the communication channels and uses them to create the procedures for receiving and transmitting information between the board and other systems such as sensors, GPS, ground control station… In FlyMaple, there are only four communication channels, which is **SerialUSB**, **Serial1**, **Serial2** and **Serial3** provided by the native drivers. As a Mapping Hardware layer, AP_HAL_FLYMAPLE provides a mapping for this situation (**fig 3-7**).

```
class HardwareSerial;
extern HardwareSerial Serial1; // Serial1 is labelled "COM1" on Flymaple pins 7 and 8
extern HardwareSerial Serial2; // Serial2 is Flymaple pins 0 and 1
extern HardwareSerial Serial3; // Serial3 is labelled "GPS" on Flymaple pins 29 and 30

static FLYMAPLEUARTDriver uartADriver(&Serial1); // AP Console and highspeed mavlink
static FLYMAPLEUARTDriver uartBDriver(&Serial2); // AP GPS connection
static FLYMAPLEUARTDriver uartCDriver(&Serial3); // Optional AP telemetry radio
```

Figure 3-7: Mapping in Harware Mapping Layer

The code in **Fig 3-7** has two parts. The first part indicates the channels of the real hardware (FlyMaple), and the second part below it presents the ways connecting uartA to uartC with these channels. The AP_HAL_FLYMAPLE acts like glue code between native drivers and abstract hardware layer.

The **Maple driver** layer shown in **Fig 3-5** includes the driver files of FlyMaple, and these files are contained in specific libraries grouped in **libmaple**. In fact, **HardwareSerial** in **figure 3-7** is a class defined in this library.

Using the multi-layers structure, ArduPilot has the benefits as well as the drawbacks of it. One of the most important advantages of the multilayer is the portability. ArduPilot is suitable not only for many types of unmanned systems, but also for many different kinds of main board. The users do not have to change from a familiar developing platform to another every time they change the board. In other words, ArduPilot is hardware independent. This advantage can also help ArduPilot to be updated with any kind of modern board without changing its basic structure. Moreover, its multilayer structure makes ArduPilot easy to modify and change for the needs of a specific user. A programmer knowing the

structure of ArduPilot can adjust it at the right level to achieve the desired targets without making many modifications.

However, for new users, ArduPilot is complex as the result of multilayer structure. ArduPilot main code has more than seven hundred thousand code lines [**18**], and a new user cannot easily find the exact place to modify the code to do something simple without spending a lot of time. Additionally, the size of the constructed firmware, because it has to contain many related information, is large comparing with other one-target code. This is a major problem for old boards, or even modern complex control systems. The code efficiency is also a minus for multi-layer structure, as it cannot use **inline** functions in many cases in which it will minimize the size of the code and increase its performance. "*When you declare a function using the **inline** keyword, the compiler copies the body of the function into the calling function, making the copied instructions a part of the calling function as if it were written that way originally. The benefit to you is a slight improvement in performance."* [**20**]. Although there are both pros and cons in using inline functions, for an autopilot systems, performance and code efficiency is very important, as they will affect the time the system need to execute each tasks, and therefore affect the response time of each task in general.

3.1.4 ArduPilot threading in FlyMaple controller board.

ArduPilot is suitable for both old and new boards; this is also a disadvantage since it cannot use all of the advantages of the new one. A simple example for this is the threading/scheduling system in ArduPilot.

At the beginning, boards did not support multithreading, so as a default, ArduPilot supports these boards with a simple timer and callbacks called a cyclic executive. The idea of this callback system is that any task that is registered in this timer system will be called at the rate of one kHz (one time each millisecond). In this one millisecond loop, each task will have a specific time to finish it mission, and then continue with the next task, sequentially. **Figure 3-8** is a description for this way of tasking.

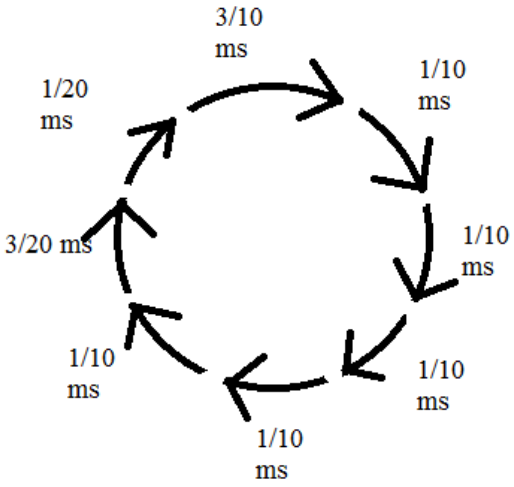


Figure 3-8: Tasking for a single thread processor in ArduPilot

Since controllers have to do a big amount of work, in fact no task is called at this one kHz rate. ArduPilot divides controller boards into two groups. The fast group (including PX4 and FlyMaple) has a MAIN_LOOP_RATE of 400 Hz, which means the fastest tasks can be called every 2.5 milliseconds. Meanwhile, slow group offers a 100Hz frequency.

For modern boards, like FlyMaple or Pixhawk, they not only can work at a much faster rate but also supports multi-threads. For these board, as described in [**21**], *"the threading approach in ArduPilot depends on the board it is built for"*. Besides the timer thread created for the single threading board as described above, at the AP-HAL abstract hardware layer, depends on the board, there are more threads such as **UART thread** for communication and **I/O thread** for tasks related to onboard memory. For example, in the **Scheduler.h** file for **AP_HAL_PX4**, as can be seen in **figure 3-9**, there are four separated threads.

Quadcopter_completed.pde | Scheduler.h | Scheduler.h

```
92      pthread_t _storage_thread_ctx;
93      pthread_t _uart_thread_ctx;
94
95      void *_timer_thread(void);
96      void *_io_thread(void);
97      void *_storage_thread(void);
98      void *_uart_thread(void);
```

Figure 3-9: Threading in PX4

In this case, four threads running and doing tasks concurrently, avoiding waste of time while polling I/O (Input/Output). For FlyMaple, however, although it is multithread supported and can use FreeRTOS for threading [**16**], ArduPilot does not fully provide these facilities. In fact, from the **Scheduler.h** file and the **Scheduler.cpp** file in **AP_HAL_FLYMAPLE**, there are no other threads besides the basic thread above.

```
void FLYMAPLEScheduler::register_timer_process(AP_HAL::MemberProc proc)
{
    for (int i = 0; i < _num_timer_procs; i++) {
        if (_timer_proc[i] == proc) {
            return;
        }
    }

    if (_num_timer_procs < FLYMAPLE_SCHEDULER_MAX_TIMER_PROCS) {
        /* this write to _timer_proc can be outside the critical section
         * because that memory won't be used until _num_timer_procs is
         * incremented. */
        _timer_proc[_num_timer_procs] = proc;
        /* _num_timer_procs is used from interrupt, and multiple bytes long. */
        noInterrupts();
        _num_timer_procs++;
        interrupts();
    }
}

void FLYMAPLEScheduler::register_io_process(AP_HAL::MemberProc k)
{
    // IO processes not supported on FLYMAPLE
}
```

Figure 3-10: FlyMaple threading in ArduPilot from Scheduler.cpp

Therefore, the only benefits of using ArduPilot for FlyMaple comparing to other older controller boards is that MAIN_LOOP_RATE of FlyMaple can run at the rate of 400Hz like PX4, which make it easier to control an autopilot system such as quadricopter. No multithreading is provided so far.

3.2 FlyMaple and ArduPilot

3.2.1 Problems

Although ArduPilot indicated an instruction to build ArduPilot code for FlyMaple [**22**], make an executable firmware for FlyMaple using ArduPilot

structure, in fact, there are still some problems between the compiled firmware and the hardware.

The first problem, as has been pointed out, is the threading and will not be discussed in this part. The next problem is more important, since it affect directly the way the embedded system communicate with other systems, in particular the Ground Control Station.

In a communication test between FlyMaple and Ground Control Station, strange phenomenon occurred, as described in **figure 3-11**. Usually, this error happens because of using a wrong version of MAVLink protocol, a communication standard for UAV which will discussed in another part of this report. However, in this case, either MAVLink Version or Baud Rate is not the problem.

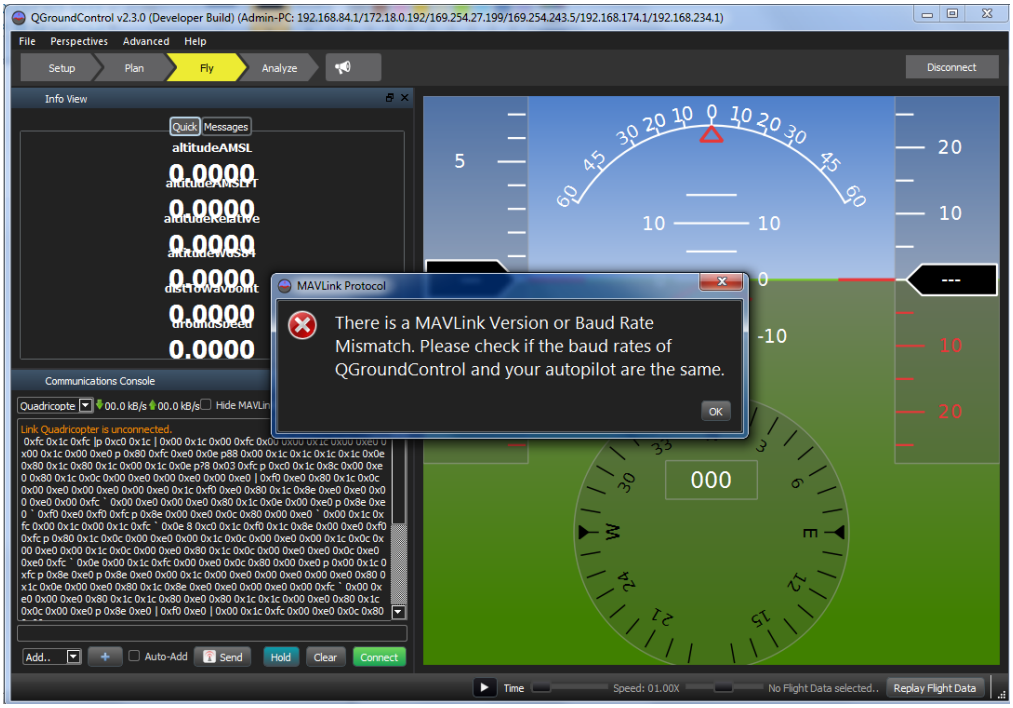


Figure 3-11: Problem with the original ArduPilot code

During testing, another problem was also noticed. As can be seen from **figure 3-12**, while communicating between two systems, many packets have been dropped, this phenomena leads to the problem of lost connection between ground control station and autopilot.

Figure 3-12: Lost packets problem

3.2.2 Solution

With the help of my supervisor, the problem has been located in the Mapping Hardware Layer, particularly in the file called **HAL_FLYMAPLE_Class.cpp** shown in **figure 3-7**. In the original code, the mapping can be described as follow

Abstract Hardware Layer	Mapping Layer	Real Hardware Driver
UartA	→	Serial1
UartB	→	Serial2
UartC	→	Serial3

Figure 3-13: Original Mapping Layer

By default, ArduPilot defines **UartA** as the serial port for Mavlink communication and console, **UartB** is used for GPS. Meanwhile, as described in the work of Nhã Đạt, for the embedded system in this project, **Serial1** of FlyMaple is for GPS and **Serial2** is used in Wi-Fi communication [**16**]. In order to fix the problems shown above, a simple modification will be made in this file. By changing two simple numbers as can be seen in **figure 3-14**, a new mapping layer has been created for the FlyMaple using in this project.

```
static FLYMAPLEUARTDriver uartADriver(&Serial2); // AP Console and highspeed mavlink
static FLYMAPLEUARTDriver uartBDriver(&Serial1); // AP GPS connection
static FLYMAPLEUARTDriver uartCDriver(&Serial3); // Optional AP telemetry radio
```

Figure 3-14: New mapping layer for this project

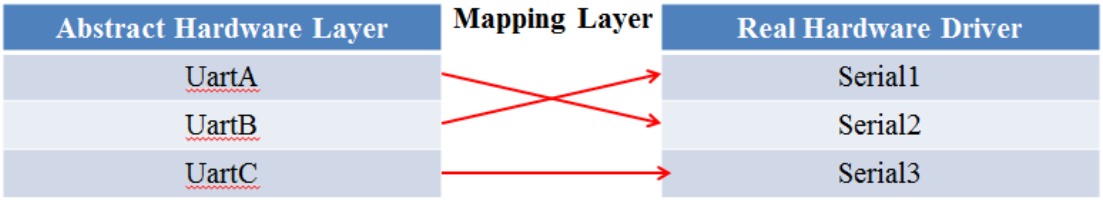


Figure 3-15: New mapping for the FlyMaple in this project

In a multilayer architecture description, the modification that has just been done is the change in the Hardware Mapping Layer. As mentioned above, this change will not affect the default settings or any other parts of ArduPilot. This is the crucial point since this modification can be done with any version of ArduPilot, which means the embedded system used in this project can continue using the newest version of ArduPilot.

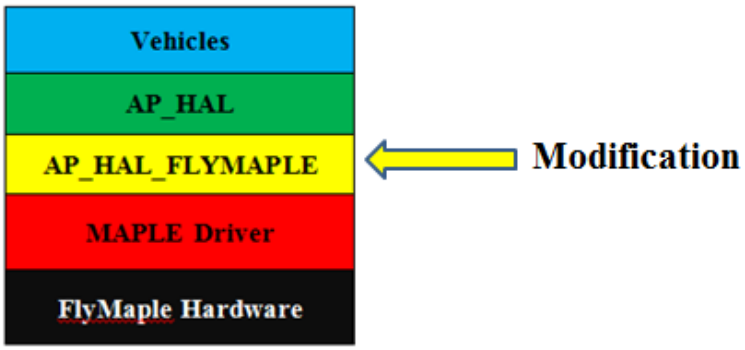


Figure 3-16: Modification for FlyMaple in this project

More importantly, this modification is suitable for the idea of a multi layered architecture application, which is adapting the mapping layer for specific hardware.

Chương 4

MAVLink protocol

MAVLink Micro Air Vehicle Communication Protocol is a common protocol for communication between Unmanned Vehicles and Ground Control Station or between Vehicles themselves. This chapter will introduce an overview about this protocol and how it is used in ArduPilot. More information about MAVLink, including its supporting functions and its checksum calculation can be seen in ***Appendix C*** *at the end of this report. After various tests and a serious evaluation, the GCS using in this project is* ***Mission Planner****, which is also a popular GCS supporting both ArduPilot and MAVLink protocol. For details about the advantages and disadvantages of this GCS comparing with other GCSs as well as the tests to evaluate each GCS, please read the* ***Appendix D****.*

4.1 Overview

4.1.1 General Communication Protocol

"MAVLink was first released early 2009 by Lorenz Meier under LGPL license" [**23**] as an effort to standardize the communication used for autopilot systems.

MAVLink has some advantages compared to other protocols, which make MAVLink one of the most popular protocols to use for autopilot communication at this time.

In order to implement this protocol into an autopilot project, the programmer just needs to add the *.h files (header files) prepared in C/C++ structs defining the meaning of messages into their project and then creates some functions to send these messages via a serial port. As long as the output messages have the right syntax, any developed Ground Control Station (GCS) using MAVLink protocol can understand them.

At this time, MAVLink provides sample messages for almost every needed situation, from basic control to camera transmitting, from calibrating system before taking action to flight plan writing and reading. Moreover, MAVLink is open-source, giving users the ability to modify its messages and to create new messages suitable for any desired purposes.

As an effort to standardize the communication of autopilot vehicles, MAVLink has been supported by many developing GCSs projects, which can be used not only in special GCS systems but also in computers, laptops and even mobile phones. Some of the GCSs for laptop will be presented in the Appendix to suggest the most suitable one for this project. Additionally, MAVLink is also the communication protocol of autopilot projects such as ArduPilot since it is lightweight, available in C/C++ language, supports creating library files for Python and C# projects, hardware independently, etc. More importantly, MAVLink does not depend on the output serial port, wire or wireless. It can handle messages from any kind of communication, UDP, TCP, Bluetooth, and even console channel, which is used in simulation such as Software In The Loop as well as Hardware In the Loop.

4.1.2 MAVLink packet anatomy.

Figure 4-1 gives the general idea of using MAVLink in an autopilot systems and Ground Control Station. The implementation of this protocol

includes two separated objects. Firstly is the integration in autopilot system. MAVLink protocol is nothing but definitions of numbers, number sequence and the order to transfer them from system to system. It does not affect the main code or the main structure of the control code, which related to calculations and control variables. While implementation the communication system, programmer will add the library files containing the meaning of each number in messages and then the procedure to handle these numbers. In C/C++, these files are *.h files as mentioned above.

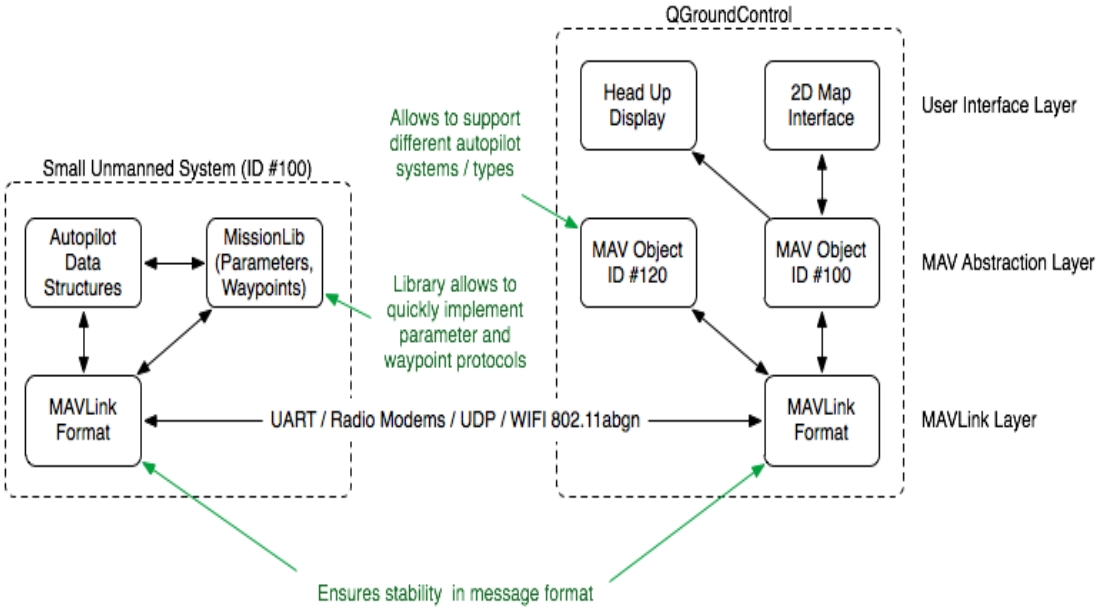


Figure 4-1: Communication between Unmanned System and GCS [24]

Secondly, it is the implementation in Ground Control Station. **Figure 4-1** introduces the architecture of QGroundControl, a GCS supporting MAVLink, and as can be seen, its structure is multilayer. This architecture provides this application the ability to control multiple different autopilot systems simultaneously. The lowest layer is the MAVLink layer with the MAVLink format and will check the correctness of the received messages, which can be transferred by any protocol. After the confirmation of this layer, the message will passed the message to the MAV abstraction layer. The system ID, which includes in each message, will separate each system with others and GCS will handle each of them separately. The highest layer includes commands and functions to interact with users.

Figure 4-2 presents the general syntax of a MAVLink message. Meanwhile, **table 2** gives more information about the figure.

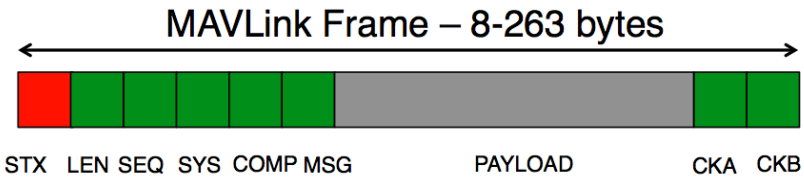


Figure 4-2: MAVLink general syntax [23]

Table 4-1: Packet Anatomy [23]

Frame	Byte Index	Content	Value	Explanation
STX	0	Packet start sign	v1.0: 0xFe v0.9: 0x55	Signal of a new package
LEN	1	Payload length	0-255	Length of the payload part
SEQ	2	Packet sequence	0-255	Number presenting the order of package, used to detect package loss
SYS	3	System ID	1-255	ID of the sending autopilot system.
COMP	4	Component ID	0-255	ID of the component of the sending system
MSG	5	Message ID	0-255	ID of the message, give the key to decode the package
PAYLOAD	6 to (n+6)	Data	(0-255) byte	Sent data
CKA	n+7	Checksum	SAE-AS4 standard [**25**]	
CKB	n+8			

Two examples for this packet anatomy can be seen in **figure 4-3** as well as the explanation of the heartbeat message in **table 3.**

```
Heartbeat message from autopilot system
FE 09 02 01 c8 00 00 00 00 00 02 03 00 03 03 18 13

Change flight mode command from GCS
FE 06 F2 FC 01 0B 03 00 00 00 01 01 7F 65
```

Figure 4-3: Some messages with MAVLink packet anatomy

Table 4-2: Detail of a heartbeat message

Group	Byte value	Frame	Explanation (decimal value)
Heading	FE	STX	Default value for MAVLink version 1.0 (254)
	09	Payload length	Payload of this message has 9 bytes
	02	Message sequence	Second message from this system
	01	System ID	Name of this system (1)
	C8	Component ID	IMU component (200)
	00	Message ID	Heartbeat message ID (0)
Data	00 00 00 00 02 03 00 03 03	Payload	- Custom mode (0 0 0 0): Preflight, not armed - Type (2) : Quadricopter - Autopilot (3): ArduPilotMega - Base mode (0) - System Status (3) : Standby - Version (3) : Version of ArduPilot
Checksum	18	CKA	Large checksum byte (24)
	13	CKB	Small checksum byte (13)

From the figures and the tables above, it can be seen that the shortest MAVLink packet has a length of 8 bytes; meanwhile the longest one can reach 263 bytes.

Moreover, by adding System ID and Component ID, MAVLink can separate not only different autopilot systems, which give it the ability to communicate with multiple systems at one time, but also different components in one system. The packet anatomy of MAVLink also provides the method to detect loss packet in transmitting, which is crucial for a GCS to check the connection with the system and for an autopilot system to trigger a failsafe procedure when it disconnects from the controller.

4.2 Utilization of MAVLink in ArduPilot

4.2.1 Tasks and files

In general, the communication with GCS in ArduPilot is implemented as in **figure 4-4**. In the main pde file of every vehicle directory, ArduCopter.pde for example, there are some tasks for communication. These tasks are put into threading as mentioned before. In this case, among the four, **gcs_send_heartbeat** and **gcs_check_input** are the most important.

A heartbeat message is a special message keeping the connection between the autopilot and GCS alive. The GCS will send signals to autopilot to check his heartbeat repeatedly and wait for its response. In case it does not receive the heartbeat, GCS will alarm "Connection lost", even though it still receives other messages from autopilot. With this alarm, the UAV will automatically trigger the "failsafe" procedure and will land down.

gcs_check_input is the listening and responding task, which is the foundation for any communication with the GCS, this is also the task where the **mavlink_parse_char** function above is called. In order to save time for more important tasks such as reading sensors or calculating rotor control signal, beside heartbeat message as mentioned above, other messages will only be sent when GCS asks for them

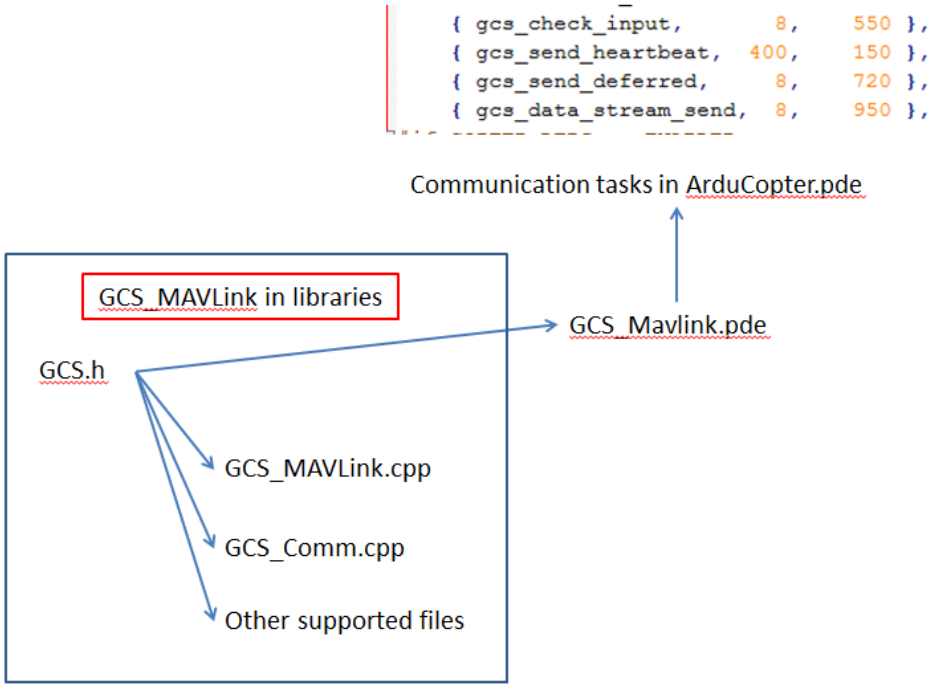


Figure 4-4: Software architecture of MAVLink in ArduPilot

The other two tasks, **gcs_send_deffered** and **gcs_data_stream_send** are supporting tasks for GCS communication. More details of the communication sequence will be discussed in the next part of this report.

ArduPilot communication with GCS is implemented by three files. **GCS_Mavlink.pde** in ArduCopter directory contains the specific commands for multicopter vehicles; two ***.cpp** files in **GCS_MAVLink** in the libraries directory include functions can be used for any kind of vehicle. Other related files such as the definition of Mavlink packages or tracking message route are also in this directory.

4.2.2 Communication procedure

4.2.2.1 General procedure

The procedure for sending and responding a message of autopilot system in ArduPilot can be described as in **figure 4-5.**

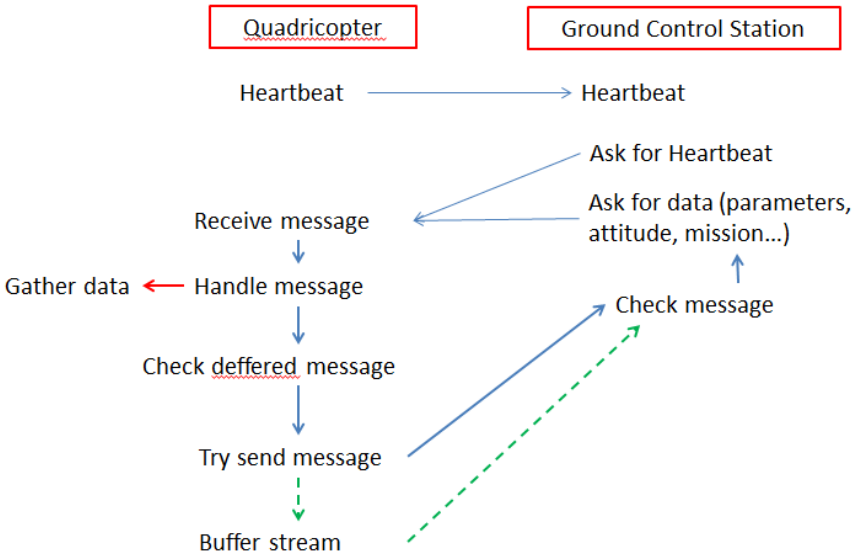


Figure 4-5: Communication procedure

As mentioned, heartbeat is a special message, therefore, it will be sent individually by the autopilot as well as will being asked regularly by the GCS. Other data will be sent when it is required by the GCS, some important data such as attitude, parameters, GPS… will also be asked repeatedly in time. Other messages like changing control values, reading/writing flight plane, flight mode… will be sent when user interact with the GCS.

After **receiving the message** from GCS, autopilot systems will **handle it and gather data** from memory or latest read values in case of sensors. Autopilot system will then **check deffered message**, a process in which autopilot will check if the message has been sent recently to prevent responding a message multiple times or if the bandwidth is full. After that, autopilot will **try to send** the message, avoiding interrupting sending important messages by lower priority messages.

In ArduPilot, there are two ways to send a message. The first way is the simple way using the basic write method of each controller board and send message as individual character. The second way is called “data stream”, which will be used in case of a big amount of data need to be sent immediately. In this method, all data will be sent in a stream all at one. For example, in a transferring control constants test between autopilot system and GCS via Wi-Fi, using data

stream will at least ten times faster than the simple way (4.86 second in data stream and 57.1 second without data stream). This duration is extremely important since when an autopilot system is activated, it cannot use all of its bandwidth just for sending and receiving parameters.

4.2.2.2 Specific procedure

With each messages, there will be a specific procedure for them. For example, **figure 4-6** describes the reading mission protocol between GCS and MAV component.

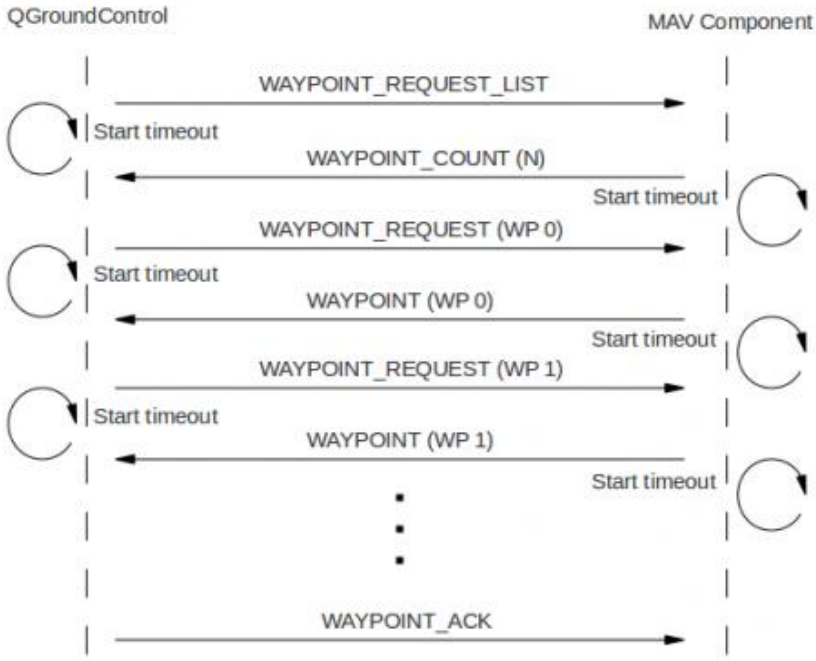


Figure 4-6: Reading mission protocol [26**]**

In this case, since many messages have to be sent and received by both components, there is an **ack message** at the end of the protocol. This is also a special message, which will be used to confirm that the protocol is finished and next protocol can be called. In some case, an ack message will contain the result of a received command.

More details about the specific protocols can be found in [**23**].

Chương 5

Navigation and Control in ArduPilot

By default, ArduPilot uses a PID controller to control the UAV in auto mode (fly using autopilot). In order to find a solution to integrate new control algorithms into this framework, a more details about the calculating procedure of ArduPilot in auto mode need to be described. In general, this procedure can be divided into two small modules: Navigation and Control. This chapter discusses about these modules and then presents the utilization of PID controller in the original code of ArduPilot.

5.1 Navigation and Control functions

5.1.1 Tasks for motors control and navigation in ArduCopter

In ArduPilot, in particular ArduCopter, the task called **motors_output()** is in charge of calculating PWM output and transmitting this signal to ESC. Since controlling the motors is one of the most important tasks, this task is put into the **fast_loop**, which will be run at the rate of 400Hz on a **FlyMaple**.

```
// Main loop - 400hz
static void fast_loop()
{

    // IMU DCM Algorithm
    // --------------------
    read_AHRS();

    // run low level rate controllers that only require IMU data
    attitude_control.rate_controller_run();

#if FRAME_CONFIG == HELI_FRAME
    update_heli_control_dynamics();
#endif //HELI_FRAME

    // send outputs to the motors library
    motors_output();

    // Inertial Nav
    // --------------------
    read_inertia();

    // run the attitude controllers
    update_flight_mode();

}
```

Figure 5-1: fast_loop in ArduPilot

The procedure of the tasks in the fast_loop as well as their functions is described in **figure 5-2.**

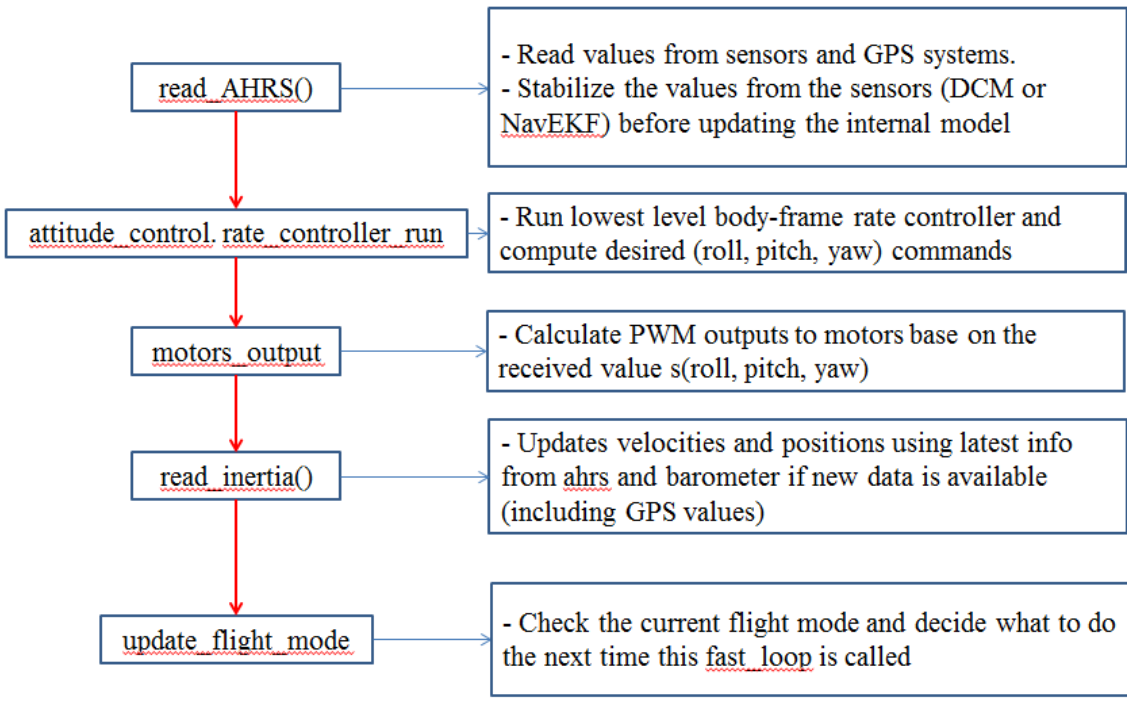


Figure 5-2: Fast_loop tasks and their function

From the figure above, there are some important notes, which are:

- Because ArduPilot supports not only the UAVs using RC but also those following a flight plan (auto mode), the most important functions in this loop are **read_AHRS**, **read_inertia** and **update_flight_mode**. In RC mode, these functions will act as the supporting systems to help the controller easily maintaining the stability of the drone. In auto mode, these functions will do everything in their own to directly control the UAV follows a planned mission.
- In fact, there are several tasks with different timer callback for sensors, GPS systems… as mentioned in the scheduling part in **chapter 3**. In other words, the acquired values from sensors or GPS system are not memorized at the same time. For instance, the control system receives the GPS messages at a frequency of 50Hz, barometer is accumulated also at 50Hz, however, compass is read only at the rate of 10Hz. The difference in calling time here affects directly the autopilot since the values used in calculation are not gained at the time. In order to overcome this problem, ArduPilot uses estimation algorithm and tries to predict the future values based on the past and the current values. These estimated values are then corrected every time the control system receives new ones.
- ArduPilot provides users with several flight modes. More details about these modes can be found in [**27**]. As a result, the calculation for PWM outputs as well as roll, pitch, yaw rates of each flight mode is different. Therefore, the inputs (position, angles) for a new control algorithm must be taken from the suitable flight modes.

5.1.2 Navigation system in auto mode

ArduPilot uses two different coordinates to navigate and calculate the required values in auto mode.

The first coordinate system, which is called earth frame in ArduPilot, is in fact the inertial frame. *"It is an earth fixed coordinate system with origin at a*

defined location. The unit vector i^i is directed the North, the j^i is to the East and the k^i is directed upward forming a clockwise trihedral with the others" [**14**]. In ArduPilot, this frame is used mostly for navigation since it can determine easily the location of a point basing on the origin. More importantly, this frame is a fixed coordinate system, which means it will not change despite of the attitude of the drone. **Figure 5-3** gives an example about this coordinate and how it is used. By default, ArduPilot uses the **Home** location as origin, therefore, **waypoint 1** in this figure will have the location as X = 0 and Y is a positive number. These values will not change while the UAV is moving unless the origin is changed.

Figure 5-3: Earth frame navigation of ArduPilot

The second coordinate is the body frame. This coordinate is fixed with the orientation of the control board, as can be seen in **figure 5-4** and **5-5.**

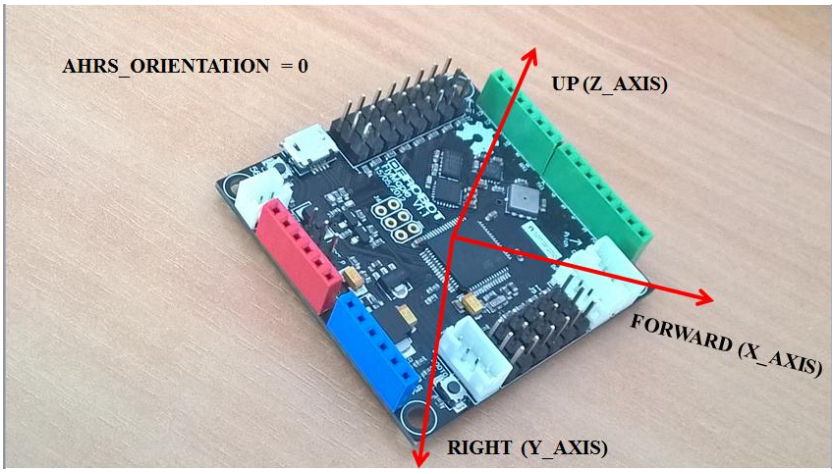


Figure 5-4: Body frame coordinate with AHRS_ORIENTATION = 0

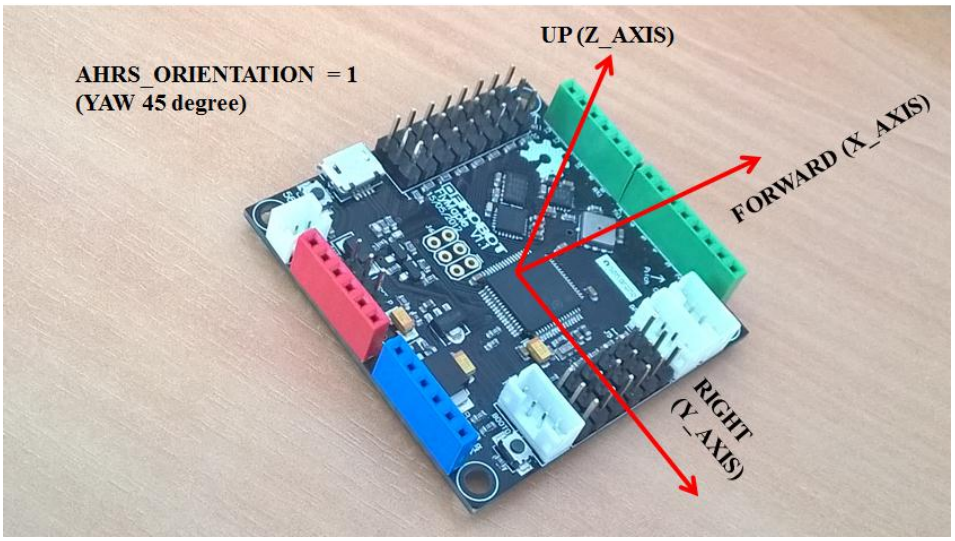


Figure 5-5: Body frame coordinate with AHRS_ORIENTATION = 1

ArduPilot uses this coordinate to calculate the PWM outputs since it reflects exactly the current attitude of the drone, make it easier to control the system.

Figure 6-6 gives a closer look in the navigation and autopilot of ArduPilot. There are five separated points used in ArduPilot auto mode:

- **Home**: As default, ArduPilot will take the launch point as Home, however, users can use GPS systems to modify this Home location.
- **Origin**: The previous waypoint which the UAV has just passed. In **figure 6-6**, Home (waypoint 0) is the Origin of the quadricopter
- **Destination**: The next waypoint in the mission. In **figure 5-6**, **waypoint 1** is the destination for the UAV.
- **Current point**: The location of the UAV at the time the system acquired it. This location can be the exact location provided by the GPS system, or it could be the estimated point as mentioned above.
- **Position target**: In order to make sure that the UAV will follow the desired path of the mission as close as possible; the path connecting Origin and Destination is divided into small segments by points called position targets. By comparing the differences between current point and position targets in earth frame coordinate, the autopilot will generate the desired values for roll, pitch, yaw, which will be used to calculated the outputs for the rotors.

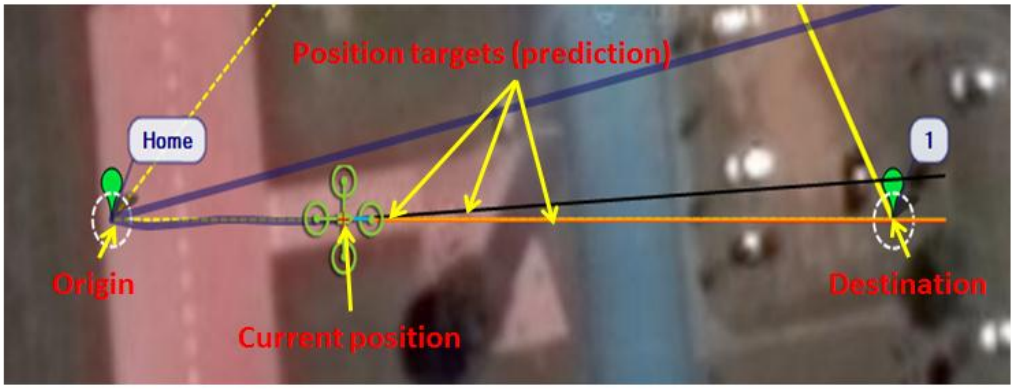


Figure 5-6: Navigation points in ArduPilot

5.2 Utilisation of PID control in ArduPilot.

By default, ArduPilot has five different PID loops to calculate the motors control signals in auto mode or to stabilize these signals in other modes. These PID control systems are considered independent from each other and each of them has their own gains as can be seen in **figure 5-7**. Among them, the PID controls related to Roll, Pitch, Yaw and Throttle will be accounted for in every calculation. The other will only take place in Loiter mode[1].

Figure 5-7: PID gains of ArduPilot

[1] Loiter Mode automatically attempts to maintain the current location, heading and altitude. The pilot may fly the copter in Loiter mode as if it were in manual. Releasing the sticks will continue to hold position. [**59**]

Another noticeable point in **figure 5-7** is that the PID gains for roll, pitch, yaw are called **Rated Roll** (P, I and D), **Rated Pitch** (P, I, and D) and **Rate Yaw** (P, I and D) respectively. This is also the idea about the PID controls of ArduPilot, using PID controller to control the **change rate** of the Roll, Pitch, and Yaw. In detail, the calculation from the beginning to the motor signals in ArduPilot can be explained as in **figure 5-8**.

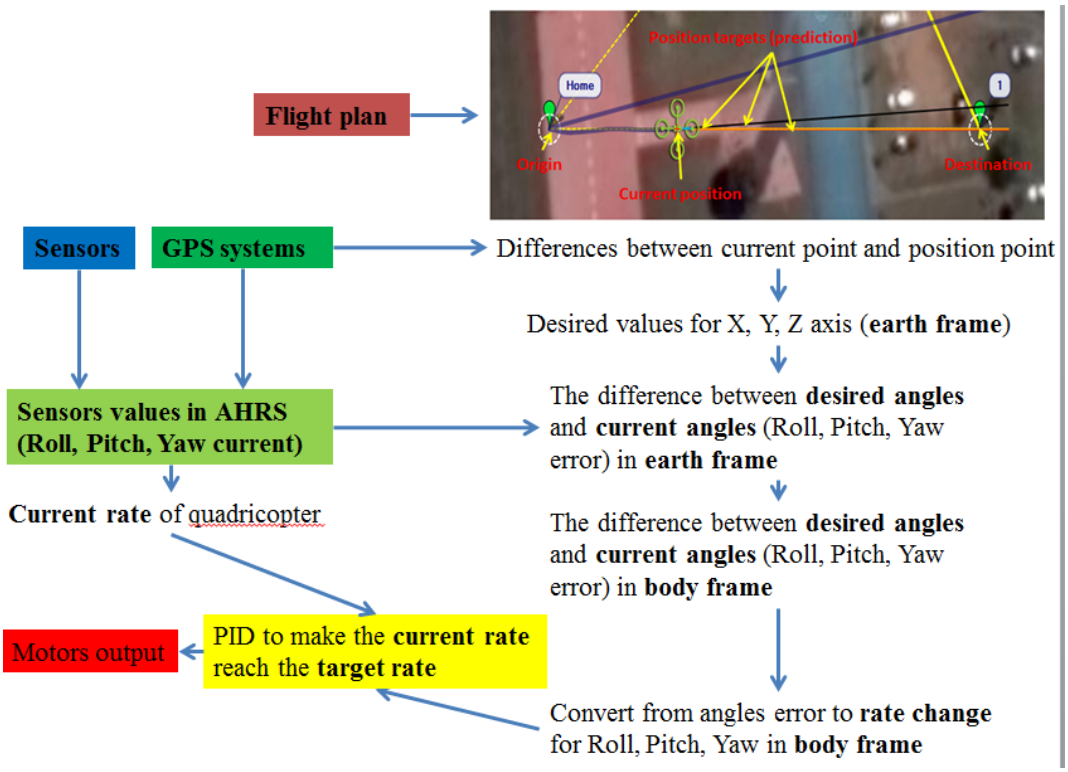


Figure 5-8: ArduPilot auto mode calculation from inputs to output

By using both coordinates, ArduPilot can use the benefits of the one to cover the other weakness. Earth frame (or inertial frame to distinguish with the real earth frame) gives the fixed position of each waypoint in relatively to origin, meanwhile, body frame takes in account the orientation and rotation of the quadricopter, makes it easier to control. The control of change rate, instead of roll, pitch, yaw directly also gives ArduPilot some advantages. These advantages will be discussed further in the introduction of new PID controller.

Chương 6

Simulation and testing using Software in the Loop method (SITL)

This part introduces the use of Software In The Loop (SITL) to simulate and test the ArduPilot code before using it in a real model. A simple compare is done to present the pros and cons of SITL in relative with other testing tools. Moreover, this part will show the modifications have to be done in order to make the simulation as close as possible with the real system.

6.1 An introduction to Software in the Loop simulation

Simulation is one of the most important steps for any design and programming procedure in general. Although it cannot describe and take in account every situation that could happen in the real world, testing a system in simulation will

help developers not only understanding the behaviors of that system in the real world but also noticing what they could do to optimize that system. For an UAV, control code will need to be tested many times in virtual world to make sure that it could work properly and prevent any unfortunate error due to control error. There are two simulation methods that are usually used, **H**ardware **i**n **t**he **L**oop (HITL) and **S**oftware **i**n **t**he **L**oop (SITL).

HITL simulation is a technique using the real hardware (an embedded system for example) with the simulated input values. By giving the control board the values that would be given to it by the real sensors in the real situation, developers could learn about the behavior of that system. The data from this simulation will help developers to predict the reacting of the same hardware in the real world in case of facing a familiar situation and then modify the control code if necessary. In short, this simulation uses **real hardware** in a **virtual environment** and learns about **real behavior** of that hardware.

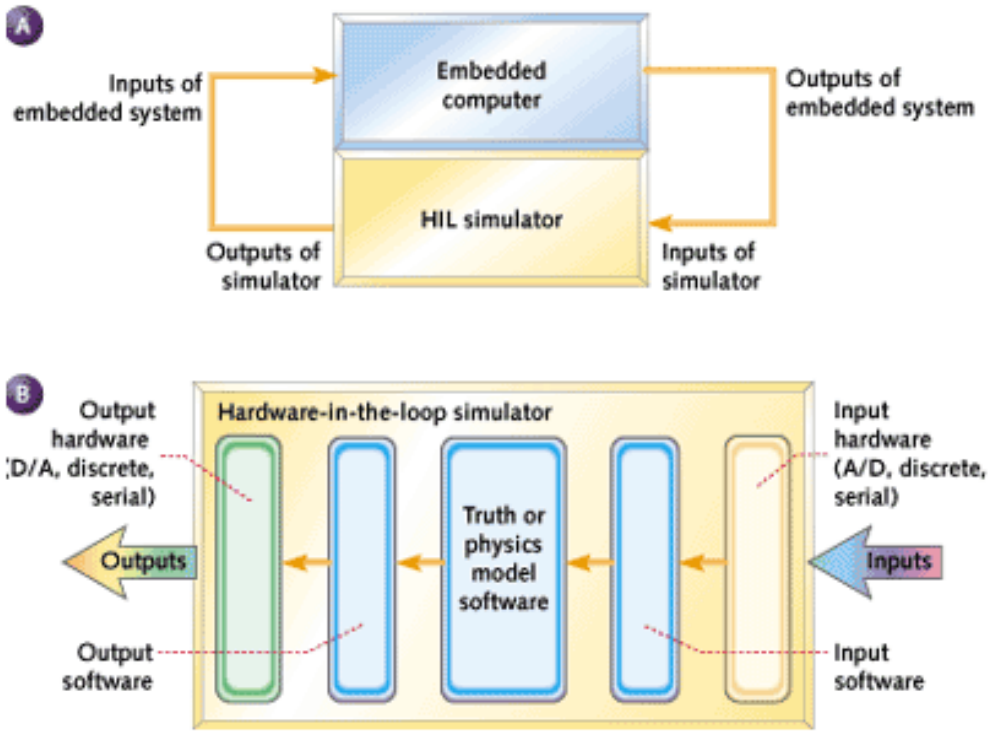


Figure 6-1: An overview about Hardware in the Loop simulation system [28]

The major difference between **SITL** and HITL, as can be seen from their name, is that SITL does not need a real hardware in the simulation process.

Indeed, with SITL, the hardware is also simulated as well as the testing environment. Both HITL and SITL have their own benefits as well as drawbacks, as can be seen from **table 6-1**.

Table 6-1: Advantages and disadvantages of HITL and SITL

	HITL	SITL
Pros	- Testing the real behavior of a real system in situations which cannot be done with real tests because of endanger. - The result is more reliable since it is the reaction of a real hardware. - More safety as well as saving more time and money comparing to real system test. - Can be done in early stage and with separated components as well as a whole system.	- Does not need a real hardware. -Saving more time and money than HITL. - Can test many different situations, some of which cannot be done in HITL. - Easier to implement.
Cons	- In some situations, creating the virtual inputs is not simple, need to understand clearly about the testing environment. - For some systems, the real hardware is not only complicated to acquire but also require a lot of space.	- The result is not as reliable as HITL since it is just the behavior of a virtual hardware. - Needs time to create as well as to validate the virtual hardware, comparing its behavior with the real one.

In fact, depending on the objectives of the test, developers will decide which simulation should be used. However, in case of ArduPilot, SITL seems to have more advantages because of two limitations of HITL:

- *It cannot run all of the autopilot code, as the low-level driver code will not see suitable inputs for a flight test when the hardware is sitting on your desk.*
- *You cannot use the sort of advanced programming tools (such as debuggers and memory checkers) that are so useful in normal C++ development.* [**29**]

Therefore, in ArduPilot, developers have created the necessary tools for users around the world to test their autopilot systems with Software in the Loop simulation.

6.2 SITL in ArduPilot

Figure 6-2 gives a simple description of SITL architecture in ArduPilot.

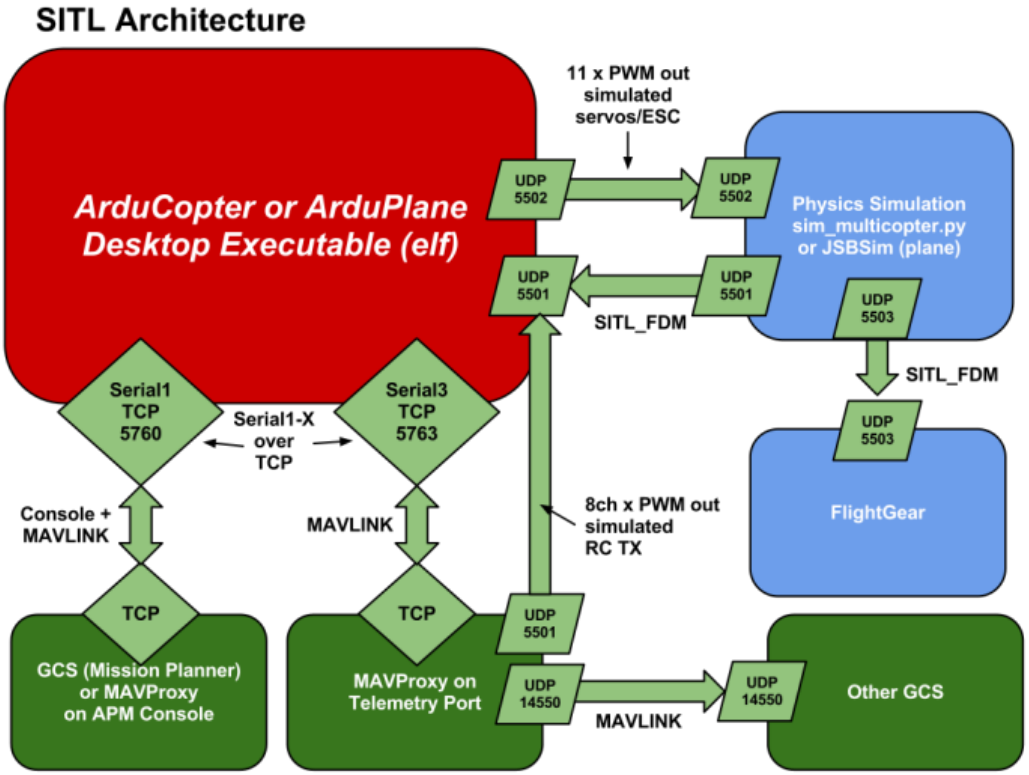


Figure 6-2: SITL architecture in ArduPilot

As can be seen, SITL architecture in ArduPilot includes three main parts. The first part is the *.elf file of the autopilot firmware built with ArduPilot. The architecture of ArduPilot as well as the process to create elf file can be found in **Appendix B**. The second part related to the Ground Control Stations and the communication between it and the autopilot system. More detail about this communication (MAVLink protocol, choosing ground control station…) can be seen in **chapter four** and **Appendix C**. In this part, only the blue boxes are the ones concerned. These boxes represent the virtual hardware and the virtual environment for SITL simulation. By putting the built firmware into this virtual

hardware, users can simulate the behavior of the hardware (in our case a quadricopter) and test the new controller.

The SITL in ArduPilot is constructed in Python, its files structure can be described as in **figure 6-3.** The files in this figure can be found in the **Tools/autotest** directory of ArduPilot

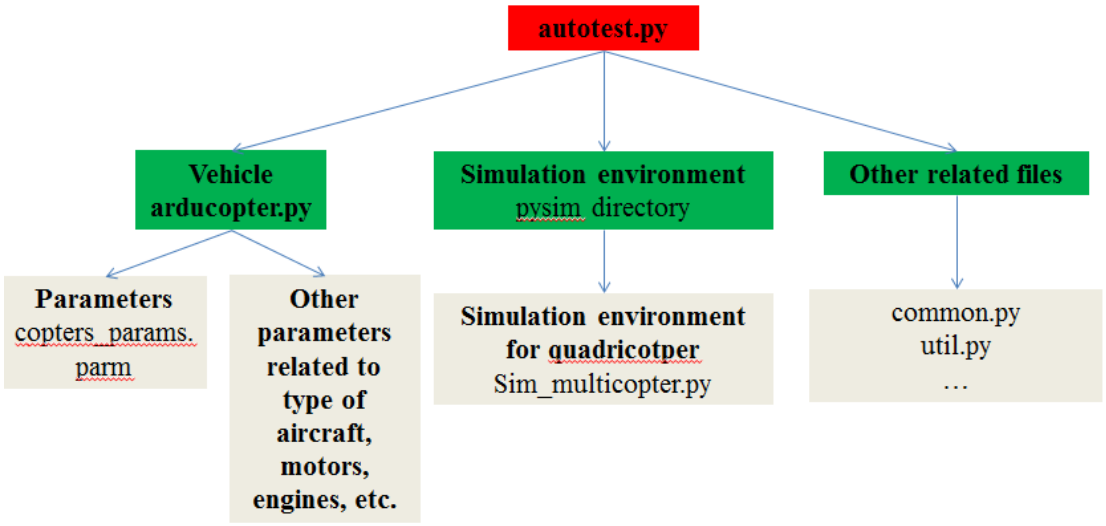


Figure 6-3: File structure of SITL simulation in ArduPilot

As can be seen from the **figure 6-3**, **autotest.py** is the main file of the simulation, which defines not only the commands that could be used to build the systems but also the simulation process. For instance, some main commands that could be used for building a simulation including in autotest.py and their functions can be seen in **table 6-2**.

Table 6-2: Some command can be used with autotest.py

Command	Function	Located file
prerequisites	checks the existence of required directories and tools to run tests	autotest.py
build.All	runs the build_all.sh script	autotest.py
build2560.ArduPlane	builds the firmware for board Mega2560 specific for ArduPlane	util.py
build.ArduPlane	builds the firmware for ArduPlane on the target board	util.py
defaults.ArduPlane	gets default parameters for ArduPlane	autotest.py
fly.ArduPlane	uses the firmware that has just been built in a SITL simulation test	arduplane.py

build2560.APMrover2	builds the firmware for board Mega2560 specific for APMrover	util.py
build.APMrover2	builds the firmware for APMrover on the target board	util.py
drive.APMrover2	uses the firmware has just been built in a SITL simulation test	apmrover2.py
build2560.ArduCopter	builds the firmware for board Mega2560 specific for ArduCopter	util.py
build.ArduCopter	builds the firmware for ArduCopter on the target board	util.py
fly.ArduCopter	uses the firmware that has just been built in a SITL simulation test	arducopter.py

In this project, our board is FlyMaple and our target vehicle is a quadricopter; therefore, the command in **figure 6-4** could be used to start the simulation process.

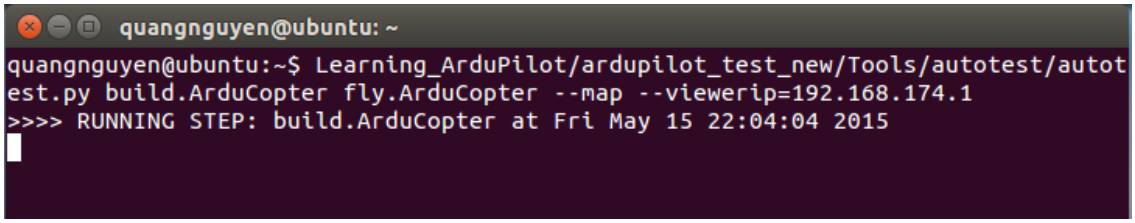
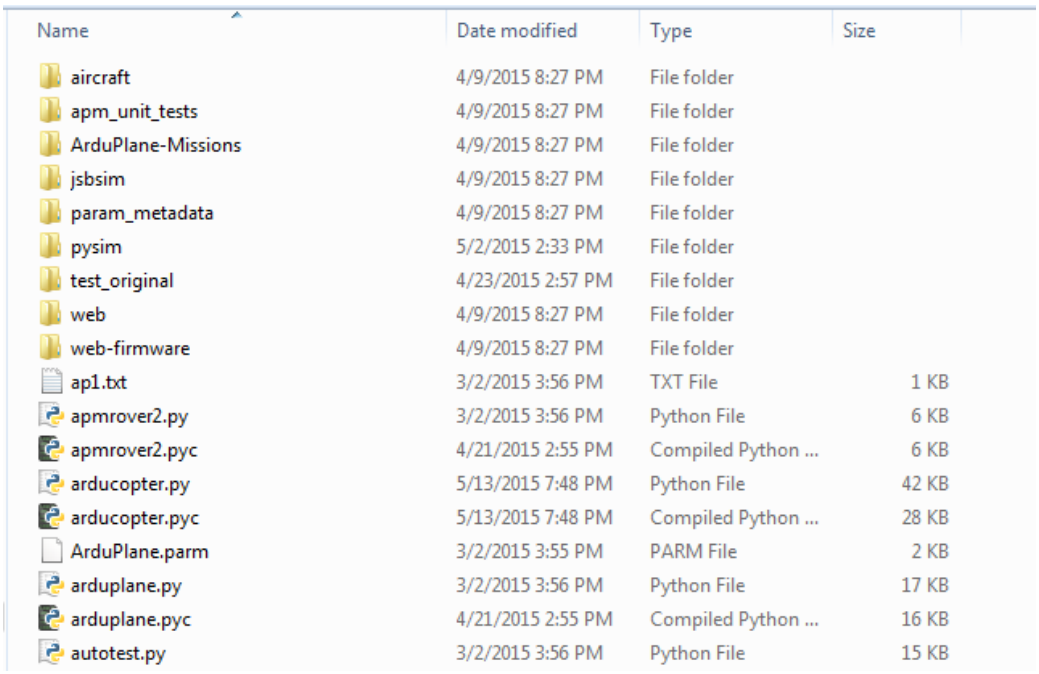


Figure 6-4: Command to start the SITL simulation

For each kind of simulation target, there are two different groups of parameter. The first group is the physical parameters of that target such as its weight, type of motor, type of propeller, etc. These values are packed in several directories such as **aircraft** or **param_metadata** and should not be modified unless users have certain knowledge about the structure of this simulation as well as about python. The second parameter group, however, is easier to modify. For ArduCopter, these values are put into **copter_params.parm** file. As can be seen in **figure 6-6**, the values in the second group are related to the values that the hardware would get during the calibrating process of a real UAV. By changing these values, or adding some, users could modify the virtual hardware in the SITL simulation to get it close to the real hardware in order to increase the reliability of the result.



Name	Date modified	Type	Size
aircraft	4/9/2015 8:27 PM	File folder	
apm_unit_tests	4/9/2015 8:27 PM	File folder	
ArduPlane-Missions	4/9/2015 8:27 PM	File folder	
jsbsim	4/9/2015 8:27 PM	File folder	
param_metadata	4/9/2015 8:27 PM	File folder	
pysim	5/2/2015 2:33 PM	File folder	
test_original	4/23/2015 2:57 PM	File folder	
web	4/9/2015 8:27 PM	File folder	
web-firmware	4/9/2015 8:27 PM	File folder	
ap1.txt	3/2/2015 3:56 PM	TXT File	1 KB
apmrover2.py	3/2/2015 3:56 PM	Python File	6 KB
apmrover2.pyc	4/21/2015 2:55 PM	Compiled Python ...	6 KB
arducopter.py	5/13/2015 7:48 PM	Python File	42 KB
arducopter.pyc	5/13/2015 7:48 PM	Compiled Python ...	28 KB
ArduPlane.parm	3/2/2015 3:55 PM	PARM File	2 KB
arduplane.py	3/2/2015 3:56 PM	Python File	17 KB
arduplane.pyc	4/21/2015 2:55 PM	Compiled Python ...	16 KB
autotest.py	3/2/2015 3:56 PM	Python File	15 KB

Figure 6-5: Some files and folders in autotest directory

```
1   FRAME        0
2   MAG_ENABLE   1
3   FS_THR_ENABLE    1
4   BATT_MONITOR     4
5   CH7_OPT          7
6   COMPASS_LEARN    0
7   COMPASS_OFS_X    5
8   COMPASS_OFS_Y    13
9   COMPASS_OFS_Z    -18
10  FENCE_RADIUS     150
11  RC1_MAX          2000.000000
12  RC1_MIN          1000.000000
```

Figure 6-6: Parameters in copter_params.parm

Besides the files mentioned above, there are some more supporting files such as the directory including the code to define the testing environment, the directory to export the result, etc. All of these files are also constructed in Python and should not be modified.

The idea about the SITL simulation of ArduPilot is that even one developer with a simple personal computer could carry on the simulation and test their new code. Therefore, in order to use the Software in the Loop simulation, a virtual machine using Ubuntu Operation System has to be set up. This virtual machine will be the environment to build and simulate the hardware. The other computer, which is the host machine, will act as the ground control station and communicate with the virtual drone via UDP connection. More details about this

setup as well as other information related to using SITL simulation for ArduPilot can be found in [**30**] and [**31**].

6.3 Modifications in the original simulation code

As mentioned above, all of the simulation code can be found in the tools/autotest directory. Although it requires many lines of code to implement a simulation for ArduPilot, developers only need to change some lines in **arducopter.py**, **multicopter.py** and **parcopter_params.parm** to set a new simulation suitable for their target.

As default, the SITL simulation for a quadricopter or ArduCopter in general includes many specific tasks. **Figure 6-7** represents some of these tasks. After being defined, these tasks will then be called alternately in the main function called **fly_ArduCopter**, which will be called directly by **autotest.py**.

```
def arm_motors(mavproxy, mav):
    '''arm motors'''
    print("Arming motors")
    mavproxy.send('switch 6\n') # stabilize mode
    wait_mode(mav, 'STABILIZE')
    mavproxy.send('rc 3 1000\n')
    mavproxy.send('rc 4 2000\n')
    mavproxy.expect('APM: ARMING MOTORS')
    mavproxy.send('rc 4 1500\n')
    mav.motors_armed_wait()
    print("MOTORS ARMED OK")
    return True

def disarm_motors(mavproxy, mav):
    '''disarm motors'''
    print("Disarming motors")
    mavproxy.send('switch 6\n') # stabilize mode
    wait_mode(mav, 'STABILIZE')
    mavproxy.send('rc 3 1000\n')
    mavproxy.send('rc 4 1000\n')
    mavproxy.expect('APM: DISARMING MOTORS')
    mavproxy.send('rc 4 1500\n')
    mav.motors_disarmed_wait()
    print("MOTORS DISARMED OK")
    return True


def takeoff(mavproxy, mav, alt_min = 30, takeoff_throttle=1700):
    '''takeoff get to 30m altitude'''
    mavproxy.send('switch 6\n') # stabilize mode
    wait_mode(mav, 'STABILIZE')
    mavproxy.send('rc 3 %u\n' % takeoff_throttle)
```

Figure 6-7: Some tasks for autopilot SITL simulation

```
    try:
        mav.wait_heartbeat()
        setup_rc(mavproxy)
        homeloc = mav.location()

        # Arm
        print("# Arm motors")
        if not arm_motors(mavproxy, mav):
            failed_test_msg = "arm_motors failed"
            print(failed_test_msg)
            failed = True

        print("# Takeoff")
        if not takeoff(mavproxy, mav, 10):
            failed_test_msg = "takeoff failed"
            print(failed_test_msg)
            failed = True

        # Fly a square in Stabilize mode
        print("#")
        print("########## Fly a square and save WPs with CH7 switch ##########")
        print("#")
        if not fly_square(mavproxy, mav):
            failed_test_msg = "fly_square failed"
            print(failed_test_msg)
            failed = True

        print("# Land")
        if not land(mavproxy, mav):
            failed_test_msg = "land failed"
            print(failed_test_msg)
            failed = True
```

Figure 6-8: Tasks are called alternately in the fly_ArduCopter command

Although a good simulation test required many tasks to find the behavior of the system in different situations, some of these default tasks are not appropriate for this project. As can be seen in **figure 6-7**, as a part of Software in the Loop simulation, the RC signals are also simulated. This is good for other simulation tests, however, in this project, RC signals will not be used to control the quadricopter in a real flight. Therefore, these tasks must be excluded from the main command **fly_ArduCopter**. For testing the new control code, fly_ArduCopter only needs two tasks, as represented in **figure 3-3**. The first task will arm the motors and the latter will make the quadricopter fly in auto mode following a specific flight plan. In order to arm the motors, users can use the RC controller or send arming command from the Ground Control Station. RC signal is simulated in the simulation to pass this procedure. Therefore, some adjustments will be done with the arm_motors task as shown in **figure 6-10**.

```
    try:
        mav.wait_heartbeat()
        setup_rc(mavproxy)
        homeloc = mav.location()

        # Arm
        print("# Arm motors")
        if not arm_motors(mavproxy, mav):
            failed_test_msg = "arm_motors failed"
            print(failed_test_msg)
            failed = True

        print("# Takeoff")


        ############# Start flight test############
        #Self writen mode
        print("#Testing Ardupilot new code");
        if not Test_mode(mavproxy,mav):
            failed_test_msg = "Test_mode failed"
            print(failed_test_msg)
            failed = True
        print("End Test")
```

Figure 6-9: Two using tasks for the test

```
30  def arm_motors(mavproxy, mav):
31      '''arm motors'''
32      print("Arming motors")
33      #mavproxy.send('switch 6\n') # stabilize mode
34      wait_mode(mav, 'STABILIZE')
35      #mavproxy.send('rc 3 1000\n')
36      #mavproxy.send('rc 4 2000\n')
37      mavproxy.expect('APM: ARMING MOTORS')
38      #mavproxy.send('rc 4 1500\n')
39      mav.motors_armed_wait()
40      print("MOTORS ARMED OK")
41      return True
```

Figure 6-10: Modifications in arm_motors task

```
def Test_mode(mavproxy,mav):
    '''Self written code for autoflight test'''
    print('You should see this\n')
    print('Waypoint is loading\n')
    # Fly mission #1
    print("# Load copter_mission")
    if not load_mission_from_file(mavproxy, mav, os.path.join(testdir, "Pitch_Roll_Test5.txt")):
        print("load copter_mission failed")
        return False
```

Figure 6-11: New Test_mode to make the Quadricopter follows a specific flight plan

The details of **Test_mode** task is shown in **figure 6-11**. In these lines, if users need to change the flight plan for testing the control code in different cases, only the name of the mission file *"Pitch_Roll_Test5.txt"* needs to be changed. After being built and implemented, the quadricopter will follow the new mission as defined in that txt file.

Beside the structure of the tasks, other configurations also have to be done if necessary to make the simulation as close as the real hardware. For example, by default, ArduCopter simulation target is a quadricopter plus frame. Depending on the real target, this frame can be changed by simply changing the settings in the arducopter.py file as shown in **figure 6-12**. In addition, the location of home, the default heading… can also be changed in the setting part of arducopter.py.

```
FRAME='+'
TARGET='sitl'
HOME=mavutil.location(-35.362938,149.165085,584,270)
AVCHOME=mavutil.location(40.072842,-105.230575,1586,0)

homeloc = None
num_wp = 0
```

Figure 6-12: Some settings related to the built target in arducopter.py

Chương 7

Solution for controller integration with ArduPilot

After understanding the basics of autopilot and navigation as well as the SITL architecture of ArduPilot, the most important part is integrate the new control algorithm into the ArduPilot framework. This chapter will include the following parts:

- *From angular speed to PWM signal*
- *The new PID controller*
- *The Integral Backstepping control algorithm*
- *Procedure to integrate new control code into ArduPilot code base*

7.1 From angular speed to PWM signal

As mentioned above, for UAV using electric motors, an ESC is installed to help the users to control the motor. From the previous work, as can be seen in [**14**],

the final output of the calculation is the angular speed Ω (rad/s). However, the control signal for the motor must be PWM as mentioned in **chapter two**. The calculation process from rad/s to PWM for each motor can be implemented; however, it is not necessary. In additional, this calculation depends on various characteristics of the real hardware such as the timer callback to send the signal to ESC, the output frequency of the ESC, etc. Since ESC can be calibrated to understand what is the minimum input as well as the maximum input it could receive, which will become the minimum throttle and maximum throttle respectively, a simple calculation procedure will be used, as shown in **figure 7-1**.

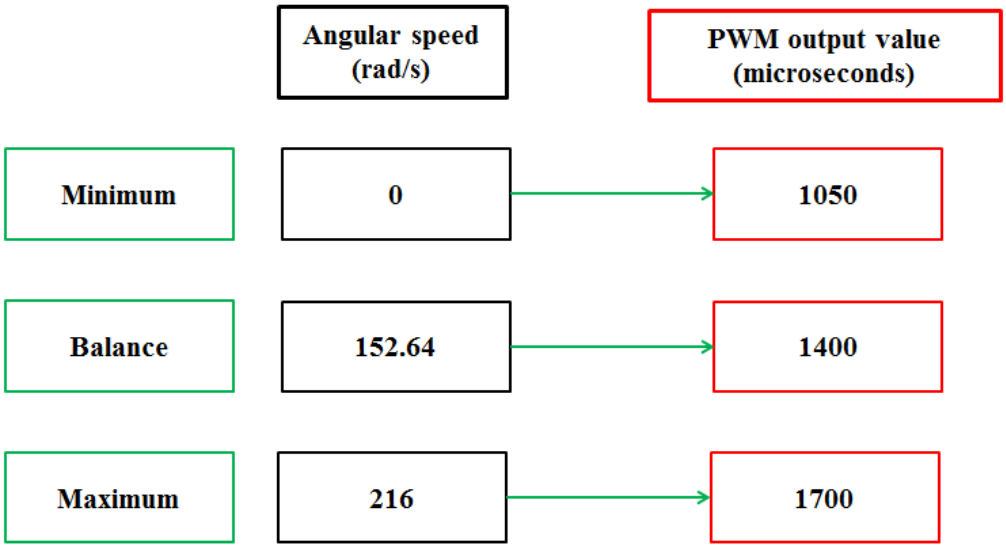


Figure 7-1: Conversion from angular speed to PWM output signal

The constant value for the angular speed of the rotors is from the previous work, as shown in [**14**]. The PWM output values are from the real system and are connected to the observed behaviors by giving different PWM values. In general, this value is between 900µs and 2000µs, depending on various things as mentioned above. With the hardware in the SITL simulation of ArduPilot, 1000 will be the minimum value and will be the disarmed value; therefore, to prevent the auto disarm, a larger value is chosen. The PWM signal at 1450 will be the value at which the quadricopter will hold its altitude, in order to make the system climbing up and down smoothly, the level value will be 1400. The maximum value can reach up to 1800 – 1900. However, if the gap between the balance output and the maximum output is too great, the quadricopter will become easily

unstable due to the sudden change every time the quadricopter reach the target altitude. Any angular speed between the minimum and balance as well as the balance and maximum will be calculated linearly. Although this solution might be an approximation, due to the fact that the system will automatically adjust its angular speed by using control method such as PID, this calculation is considered the easiest way to convert from angular speed to PWM output signal.

Nevertheless, more test and calculation must be done for the real quadricopter, in particular determining the balance value for PWM control signal. The more precise this value, the more stability the real system should be.

7.2 New PID control algorithm

The PID controller described in this report is the result from the previous work. More details about the equations and models of this PID controller as well as other controllers can be found in [**16**] and [**14**]. From the model expressed in Simulink, using Gene-auto[1], a C/C++ code structure has been created and is ready to be integrated into ArduPilot.

Comparing between this new PID controller and the original one of ArduPilot, there is one major difference: the control target. The original controller controls the **change rate of Roll, Pitch, Yaw** as mentioned above. Meanwhile, the new PID controller **controls directly the Roll, Pitch, Yaw angle** of the quadricopter. There are some advantages along with the disadvantages for this change.

The most important advantage of the original controller is that in some certain cases, it prevents the overshoot better than the new one. In case of a large change

[1] *Gene-Auto is an open-source toolset for real-time embedded systems. The toolset takes as input a functional description of an application specified in a high-level modelling language (Simulink/Stateflow/Scicos) and produces C code as output.* ***Invalid source specified.***

of command, for example changing heading from North (0 yaw degree in earth frame) to South (180 yaw degree), the new PID controller might have big overshoot, even though it also depends on the gains. The process of both controllers for this example are described in **figure 7-2.**

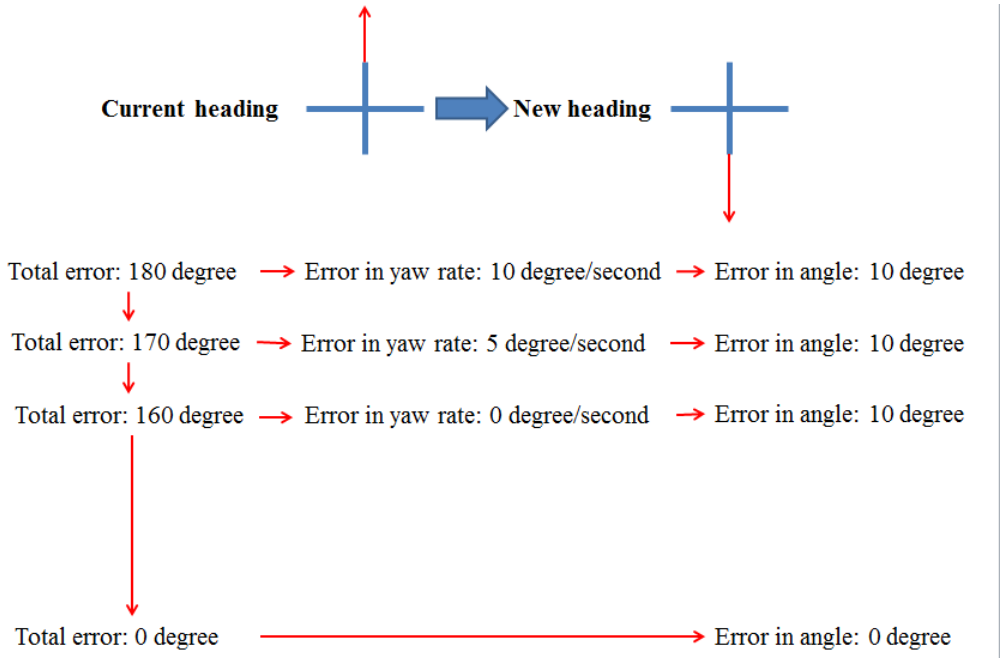


Figure 7-2: Overshoot happens with the new PID controller

At the beginning, there is no yaw change rate since the quadricopter is stable and is holding it attitude. The error of yaw rate and error of yaw angle is the first line. With this difference, PID controller will change the motor outputs and create the difference between motors to make the quadricopter rotate around its z-axis. In the second loop, because now the quadricopter has started to rotate, the error of yaw rate is now smaller than in the first loop, this difference will once again calculate with the PID. After a few loops, the current yaw rate will reach the target yaw rate defined by the developer. The system yaw rate is now stabilized. Meanwhile, in case of using the PID to control directly the yaw, the error of yaw angle always meet its maximum value, which is set to prevent the system rotates to fast. This maximum error is read again and again, as a result, the PID controller will try to increase the rotate speed. Moreover, the saved error of the system will also increase. These phenomenons will lead to two consequences. The first one is that now the rotating speed will increase

continuously, when the heading finally reaches 180 degree, the rotation speed is now too fast, it will take more time to slow down, change the rotate direction and then come back to the desired heading. Secondly, since in PID controller, the error from the past is always a part of the calculation for the current output value, if the error becomes too big, while the heading finally reach South and then past it, the controller will not respond fast enough for this change. Consequently, overshoot will happen as well as settling time will be lengthened. The problem described above will also happen in case of descending or ascending, which requires not only the precisely but also the fast response.

However, the new PID controller also has some advantages. Since this controller does not depend on the yaw rate, which is usually measure by an accelerometer, it will neglect unwanted system error of this sensor. In addition, the new PID controller will reduce the amount of calculation, which will save some time for the controller to handle other tasks. This benefit is essential for both old controller boards with slow calculation speed and new ones with high main loop frequency. The inputs of the this PID controller are presented in **table 7-1**.

Table 7-1: Inputs of the new PID control algorithm

Inputs	Unit
Error between current altitude and desired altitude	meter
Error between current roll and the desired roll	radian
Error between current pitch and the desired pitch	radian
Error between current yaw and the desired yaw	radian
Current roll	radian
Current pitch	radian
Time since the board is powered	second

7.3 Integral Backstepping control algorithm

"Backstepping control is a rescursive algorithm that breaks down the controller into steps and progressively stabilizes each system" [**32**]. This control algorithm

can be divided into three subsystems, which are Altitude control, Position control and Altitude control, each of them has a specific task and working together to control not only the stability of the system but also make it move following a desired path. This controller works well with the dynamic of the quadricopter [**33**], however, it is dynamic unstable with the disturbances [**32**]. In order to increase its robustness, an Integrator has been added into the controller; make it become the Integral Backstepping controller in **figure 7-3**. More information about this controller and it Simulink model can be find in [**14**].

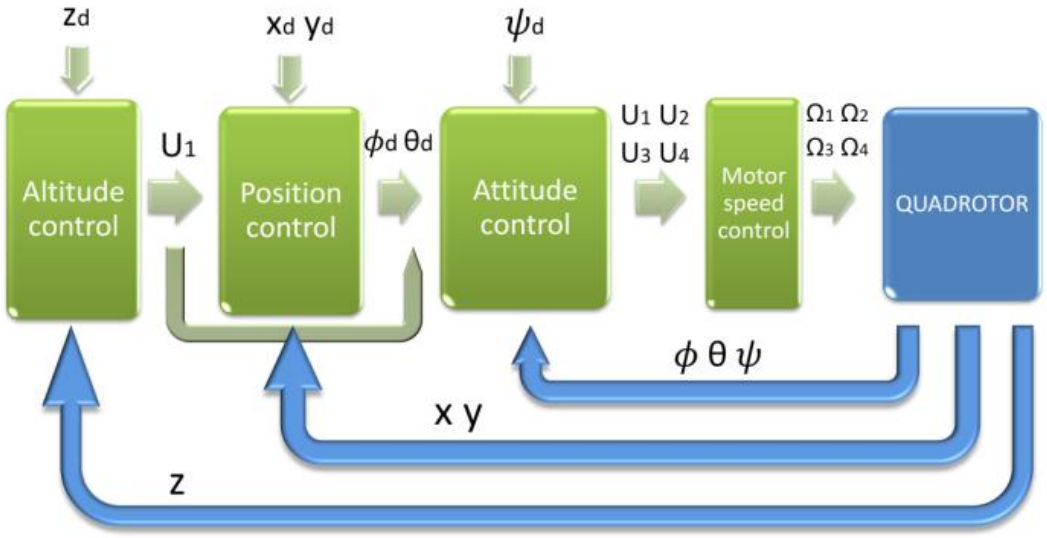


Figure 7-3: Integral Backstepping control algorithm [14]

In order to use Gene-auto to create the C/C++ code and to integrate this controller into ArduPilot, some modifications have to be done. One of the main points of the modifications is discretizing, which will not only create the suitable blocks for Gene-auto but also make the system closer to the behavior of the real system. **Figure 7-4** and **figure 7-5** presents a part of the **X_Position_control**, a subsystem in the **Position control** block before and after adaptive the model for Gene-auto. This work is the result of another internship.

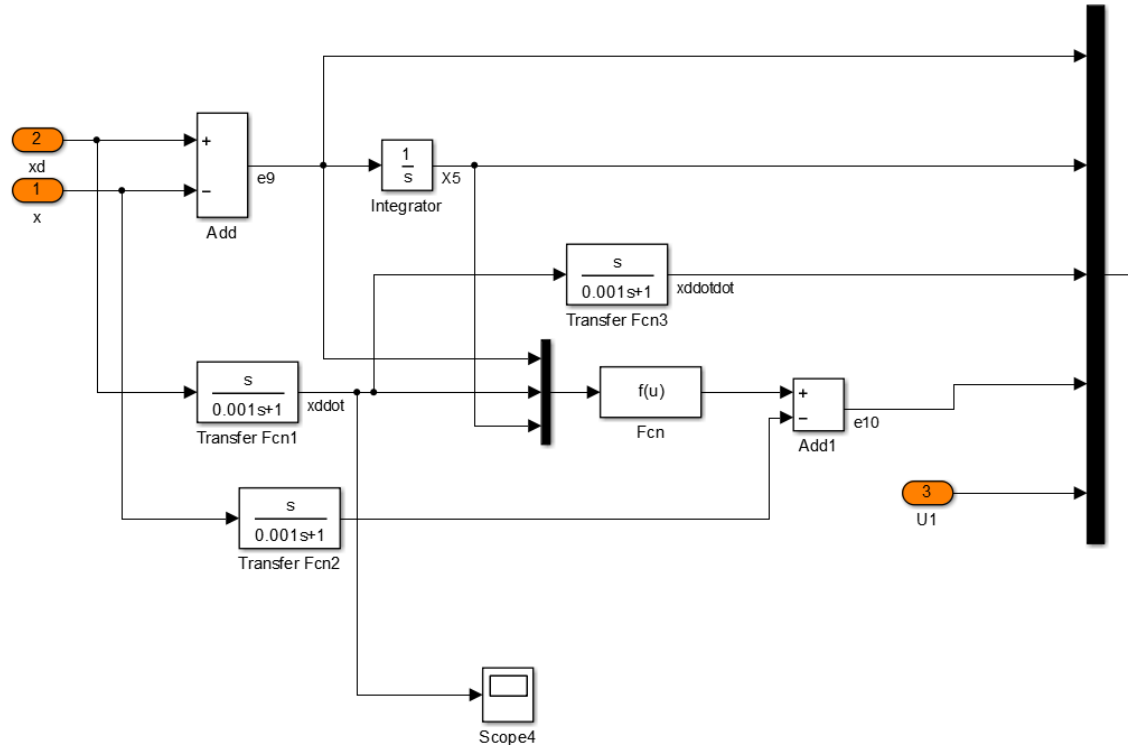


Figure 7-4: Original Simulink model

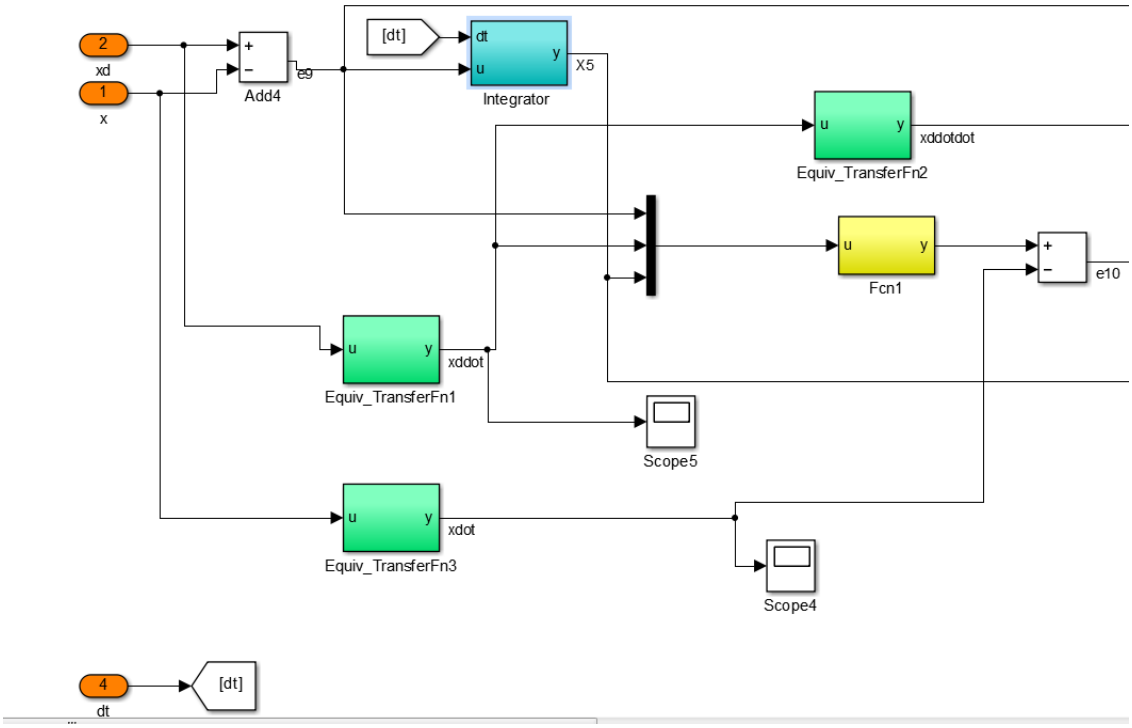


Figure 7-5: Discretized model

From the idea of the Integral Backstepping control algorithm and the Simulink model, in fact there are two different solutions for integrating this controller into ArduPilot. The first solution is using the completed Simulink model with the Altitude, Position and Attitude Control block. Meanwhile, since the **Position**

blocks is used only to generate the desired values of ϕ_d and θ_d, which is basically has been done very well in the navigation system of the ArduPilot framework, the second solution will be a combination between the IB controller and the facilities of ArduPilot framework, as can be seen in **figure 7-6**.

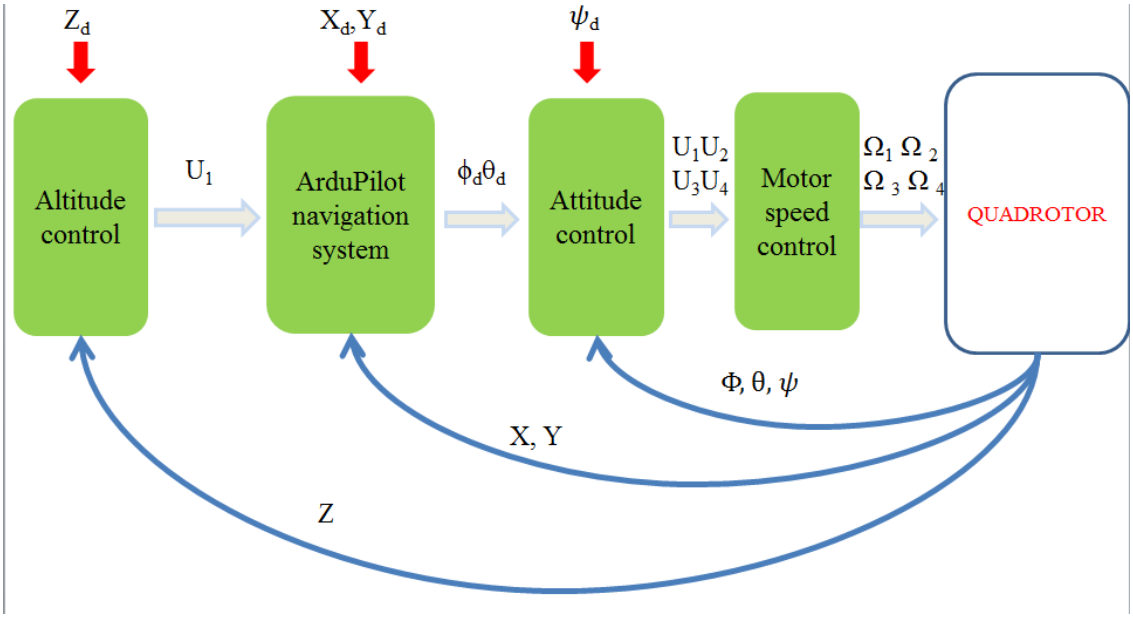


Figure 7-6: Simple IB Controller combines the IB Controller with the Navigation of ArduPilot

Each solution above has the benefits as well as drawbacks. With the combination one, the obvious cons is that it is not exactly the controller of the previous project. Since the navigation system of ArduPilot might differ from the Position block of the original IB control algorithm, this controller is not a completed form of the IB controller. On the other hand, this modified version has some advantages. By using the existent navigation system of ArduPilot, this controller can approach the facilities provided by the framework, helping it to reduce not only the amount of calculation by almost fifty percent but also the amount of works to generate and modified the Position blocks for Gene-auto. More importantly, this simplified IB controller has more connection with the original framework than the other, making it easier change the control variables such as the airspeed or waypoint radius via MAVLink protocol.

With the pros and cons of each controller, from the Simulink model of the Integral Backstepping control algorithm, this project will create two controllers.

The first one is called **"Simple IB Controller"** which is the combination mentioned above between the Simulink model and the navigation of ArduPilot. The second one is the fully IB controller from the Simulink called **"IB Controller"**. List of inputs for the simple IB controller and the IB controller can be found in **table 7-2** and **table 7-3** respectively.

Table 7-2: Inputs for the simple IB controller

Inputs	Symbol	Unit
Current altitude	z	meter
Desired altitude	z_d	radian
Current angles (Roll, Pitch, Yaw)	ϕ, θ, ψ	radian
Change rate (Roll, Pitch, Yaw)	$\dot{\phi}, \dot{\theta}, \dot{\psi}$	radian/s/s
Desired angles (Roll, Pitch, Yaw)	ϕ_d, θ_d, ψ_d	radian
Time since the last time this control algorithm is called	dt	second

Table 7-3: Inputs for the IB controller

Inputs	Symbol	Unit
Current altitude	z	meter
Desired altitude	z_d	meter
Current angles (Roll, Pitch, Yaw)	ϕ, θ, ψ	radian
Current position (in earth frame with the origin at Hone position)	x, y	meter
Desired position (in earth frame with the origin at Hone position)	x_d, y_d	meter
Change rate (Roll, Pitch, Yaw)	$\dot{\phi}\dot{\theta}\dot{\psi}$	radian/s/s
Desired Yaw	ψ_d	radian
Time since the board is powered	t	second

7.4 New module to integrate new controller into ArduPilot

Figure 7-7 introduces the general idea of this module.

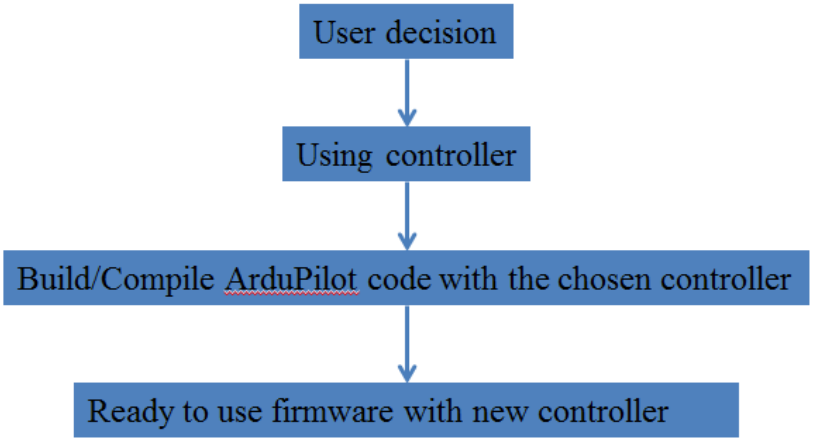


Figure 7-7: How the new module should work

The most important idea about this module is that users can change the embedded controllers with just a simple switch. This solution will not only reduce the amount of future work while integrating new controllers but also simplify the controller integrating procedure, make it easier for developers to test and conclude the pros and cons of each controller.

Figure 7-8 gives an introduction about the generated module to integrate new controller into ArduPilot.

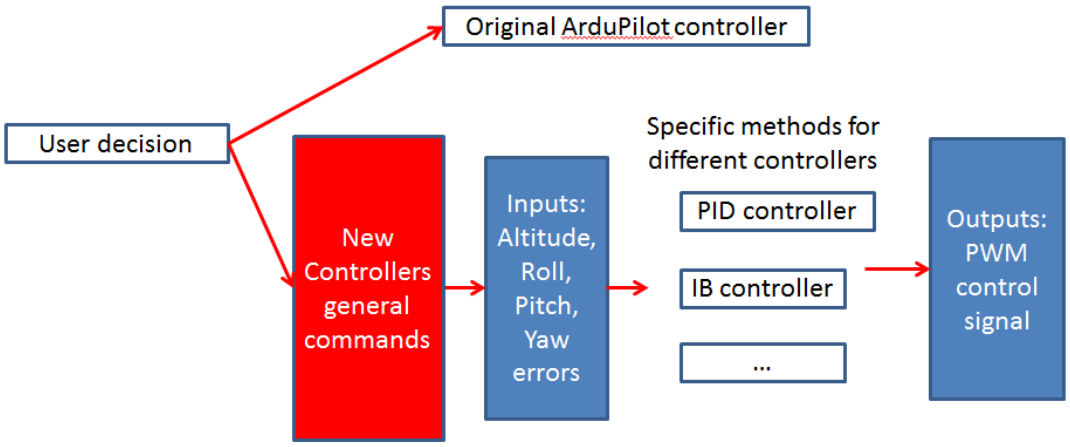


Figure 7-8: Overview of the controller-integrating module

Because of the differences of ArduPilot original controller comparing with other controllers generated in the previous work of this project, while users decide to use the original code, this module will be neglected and the built firmware will work as the original ArduPilot. In other case, for example, the decision is using the new PID controller; this module will redirect the calculation process as

shown above. Since the inputs (altitude, roll, pitch, yaw error) and outputs (PWM signal to control the motors) of new controllers are the same, the general commands will be used to get the inputs as well as send the outputs to motors.

7.4.1 Implementation of the generated module

There are several solutions to implement this controller-integrating module. One of the solutions is the multi-layered structure. Multi-layer structure, as presented in the section about ArduPilot, is one of the most practical ways when developers need to implement a multiple targets application. As shown in **figure 7-9**, just by generating the code for the virtual controller class, developers can make the firmware for many other real controllers and the used controller can be changed just by a simple change.

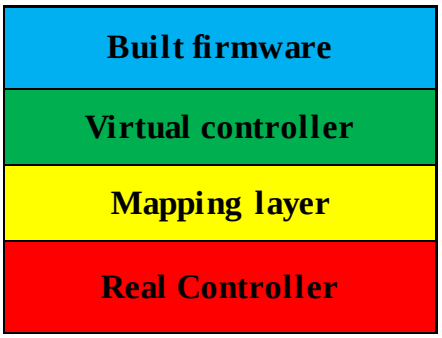


Figure 7-9: Multilayer structure to switch between controllers

This multi-layer architecture has been successfully implemented into ArduPilot, which makes this framework become one of the most popular frameworks for compiling autopilot firmware for drones.

However, this solution has some drawbacks. The complexity of the code structure is the major disadvantage. In order to use this multi-layer structure, many lines of code need to be modified, not only for the virtual class but also for the mapping layers. Due to its complication, this architecture is also not very friendly for new users, some of which are not familiar with C/C++ code. More importantly, this solution requires many modifications each time a controller needs to be integrated into ArduPilot. Since this solution require the mapping layer for each controller, developers will need to do more than just adding new controllers in C/C++ code and then use them directly. Although this multi-layer

structure will have many benefits as shown in **chapter three**, for a simple module like this, it is not very effective.

Another way to generate this module, which is the chosen one in this project, is using "switch" and "if" command. This solution is also based on the multilayer architecture, which is *"general commands fits all controllers"*, however it is simpler. Because of its simplicity, it not only reduces the effects of this new module to ArduPilot code structure but also makes it easier for future work to integrate other controllers. Additionally, it requires just basic knowledge about C/C++ of developers, unlike the multi-layer architecture above requiring a high understanding in the structure of C/C++. The procedure of creating and embedding the controller-integrating module into ArduPilot framework includes the following steps:

- Create a ***.h** file in the ArduCopter directory containing the controllers that can be used. Until now, there are only three controllers, the default controller of ArduPilot, the new PID controller and the simple IB controller.

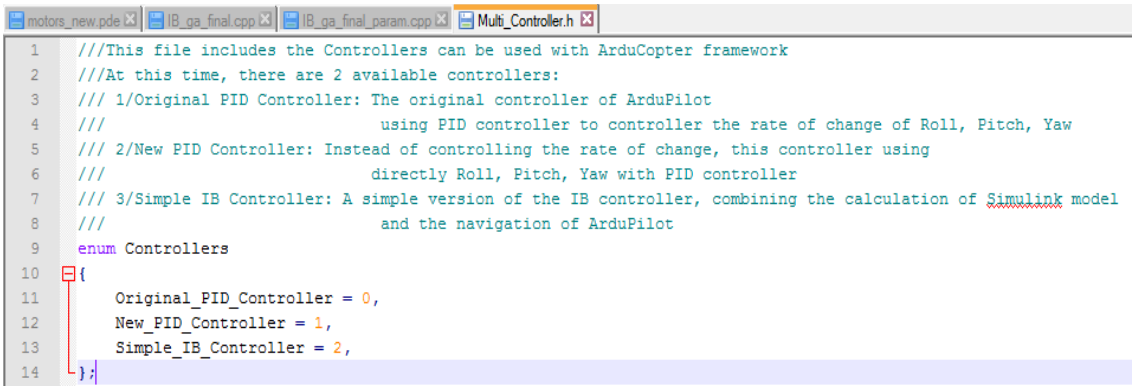


Figure 7-10: A header file defines the name for different controllers

- Modify code in ArduCopter.pde as shown in **figure 7-11**. Depending on **using_controller**, the module will change the calculation process as described in **figure 7-10**.

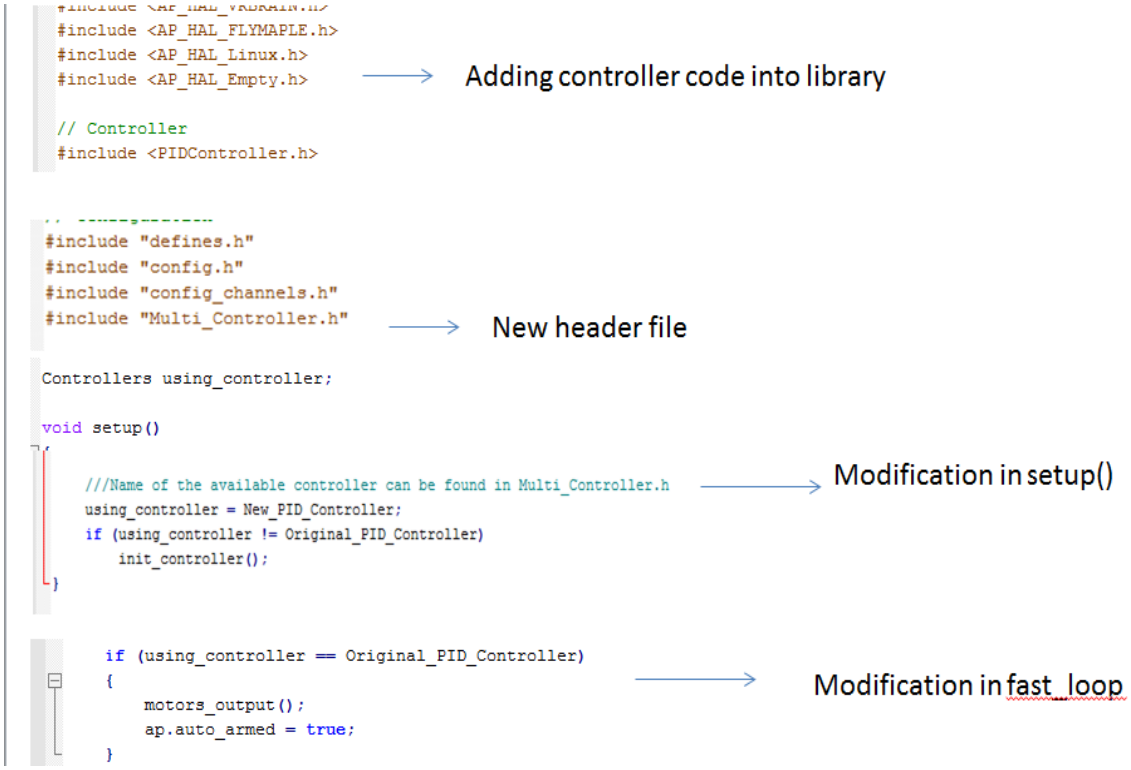


Figure 7-11: Some changes in ArduCopter.pde

- Create the general commands for the new controllers. There are three of them that need to be implemented: **init_controller** defines the initial values; **inputs_to_outputs** gets the required inputs, parses them then send the outputs to the motors calculation; **motors_output** converts outputs of the new controller into angular speed rad/s and send them to the PWM motors calculation. All of these methods will be put into a new pde file in the ArduCopter directory. **Figure 7-12**, **fig 7-13** and **fig 7-14** shows the code of each method respectively.

```
///Define variables for each controller
#if using_controller == New_PID_Controller
//IO structure
t_PIDcontroller_io io;
// State structure
t_PIDcontroller_state state;
#endif
```

→ Define the required types for each controller

```
///Initial command for each controller
static inline void init_controller()
{
    switch (using_controller)
    {
        case Original_PID_Controller:
            break;

        case New_PID_Controller:
            PIDcontroller_init(&state);

            break;
    }


}
```

→ Call the proper init command for each controller

Figure 7-12: Initial code for each controller

```
///get inputs from the auto mode and then pass it into controllers
void inputs_to_outputs(float z_error, Vector3f output, float roll, float pitch)
{
    if (using_controller != Original_PID_Controller)
    {

        GAREAL *rad_per_second;
        //Integrate the calculation for new controllers here
        switch (using_controller)
        {
            case Original_PID_Controller:
                break;

            case New_PID_Controller:
                rad_per_second = PID_calculate(z_error, output);

                break;
        }
        motors_output(rad_per_second, roll, pitch);
    }
    else return;


}
```

Figure 7-13: From the general command inputs-outputs command to the specific one for each controller

```
/// motors_output - send output to motors library which will adjust and send to ESCs and servos
void motors_output(GAREAL *output_value, float roll, float pitch)
{
    // Limits for our quadrotor
    float b = 0.00012; //Ns2
    float d = 0.000003; //Nms2
    float l = 0.225; //m
    float m = 1.14; //kg
    float Torque;
    if ((cos(roll)*cos(pitch)) != 0.0)
    {
        Torque = (output_value[0] + 9.81)*m/(cos(roll)*cos(pitch));
    }
    else Torque = (output_value[0] + 9.81)*m/(cos(roll)*cos(pitch)+0.001);

    //Constrain for io.y
    Torque = constrain_float(Torque, 0, 22.34);
    output_value[1] = constrain_float(output_value[1], -1.257, 1.257);
    output_value[2] = constrain_float(output_value[2], -1.257, 1.257);
    output_value[3] = constrain_float(output_value[3], -0.279, 0.279);
```

Figure 7-14: From outputs of controller to PWM signals (some of the code is removed)

- Modify the original code of ArduPilot to get the inputs.

Some other modifications have to be done to access the inputs (altitude, roll, pitch, yaw error, etc.). Since the navigation and estimation to get the desired values is different with each flight mode, in order to get the right inputs for new controllers, the flight mode code will need to be changed, as described in the following figures.

```
        case AUTO:
        if (using_controller == Original_PID_Controller)
            auto_run();
        else
            auto_run_multicontroller();
            break;

        case CIRCLE:
            circle_run();
            break;

        case LOITER:
            loiter_run();
            break;

        case GUIDED:
            guided_run();
            break;

        case LAND:
        if (using_controller == Original_PID_Controller)
            land_run();
        else
            new_land_run();
```

Figure 7-15: Modifications in flight_mode.pde

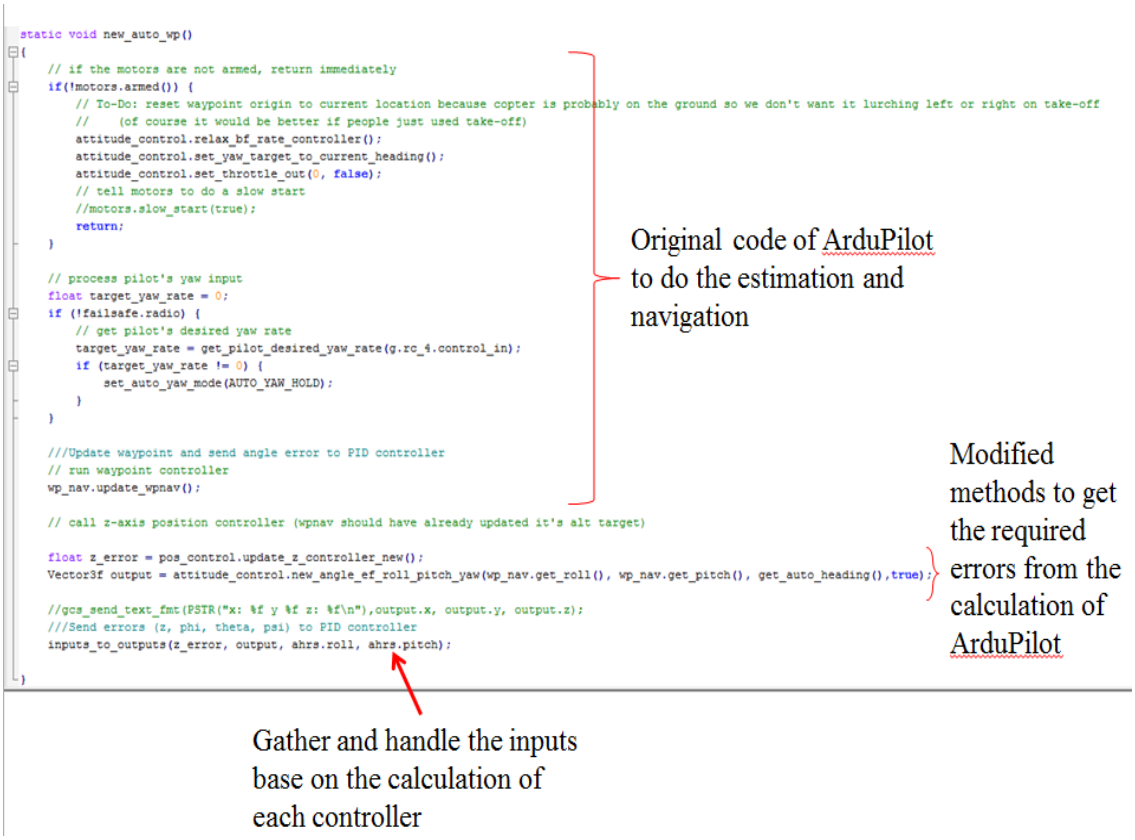


Figure 7-16: A simple modified flight mode to use new controllers

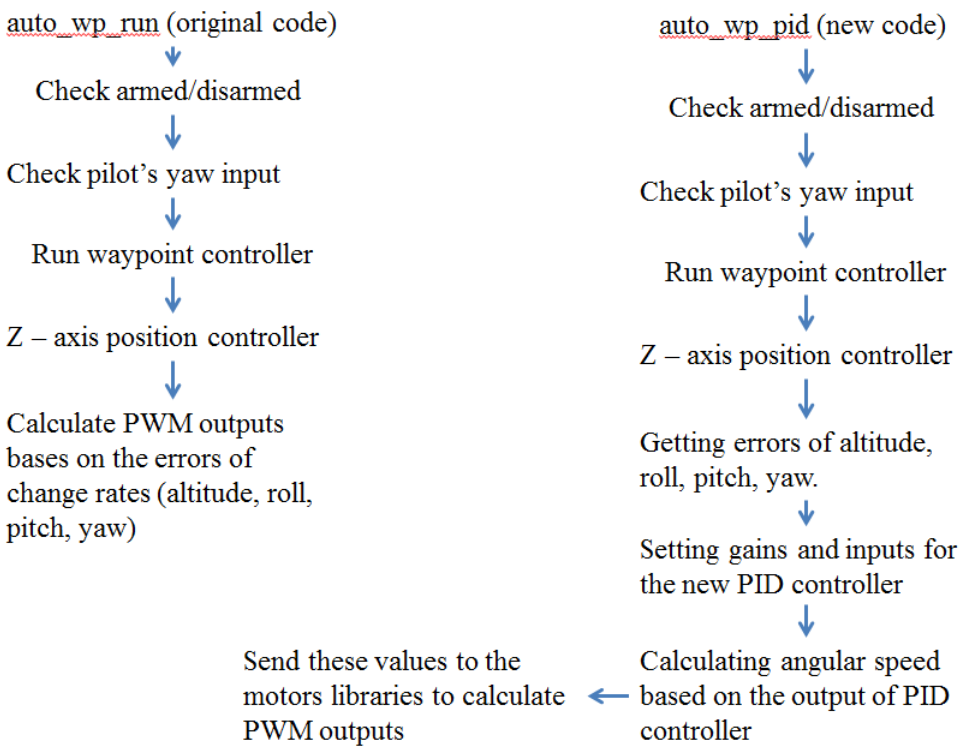


Figure 7-17: Old command (left) and new one (right) with the tasks they will carry on

- Create a conversion method to transfer from angular speed into PWM signal. The idea of this command can be found at the beginning of this part.

After finishing the above steps, a new module to integrate new controllers into ArduPilot has been created as well as fully embedded. The work from now on will be more simply, which is adding new controllers into this module and testing them with the simulation or real flight tests.

7.4.2 Adding new control algorithm with generated module

In order to integrate a new controller into the described module, developers should follow the below procedure:

- Generate C/C++ code for the control algorithm and put the code into the library directory of ArduPilot. These code files can be programmed manually or using a support tools to generate the code from a control model such as Gene-auto or the Code generator of Matlab.

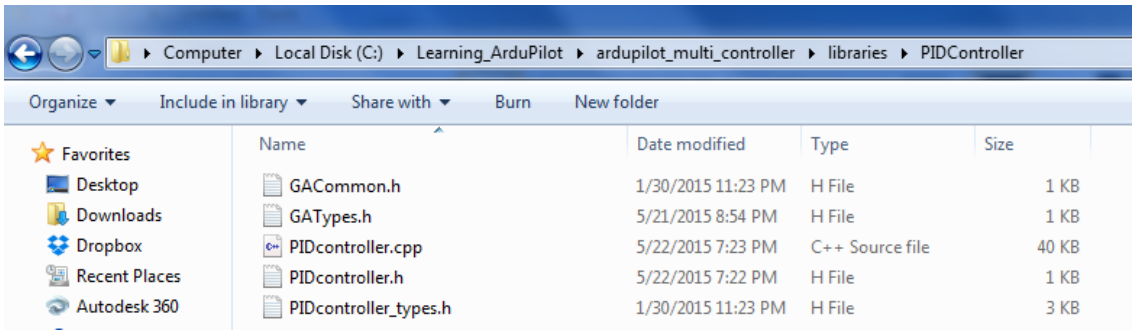


Figure 7-18: C/C++ code of the PID controller in the library of ArduPilot

- Add the name of this controller in the **Multi_Controller.h** file as in **figure 7-19.**

```
///                                    directly
enum Controllers
{
    Original_PID_Controller = 0,
    New_PID_Controller = 1
};
```

Figure 7-19: Decide the name to call the new Controller

- Change the using_controller in the main code of ArduCopter.pde

```
///Name of the available controller can be found in Multi_Controller.h
using_controller = New_PID_Controller;
```

Figure 7-20: Choose the controller

- Add the specific methods of the new controller into the structure of the general commands as described in **figure 7-12** and **fig 7-13** above.

The new controller has become a part of the controller-integrating module. By switching the **using_controller** in the main code, developers can compile and build the new firmware using any embedded controller. As can be seen, although the procedure creating the tool to integrate the controllers is complicate, the utilization of the new tool is remarkable since it has minimized the amount of work to integrate new controllers in the future. Moreover, the result about switching the controller around with just a simple variable is very impressive since with just a simple modification with the MAVLink protocol, users can change the control algorithm instantly. This achievement will be crucial in case of testing and comparing new controllers with the others as it would save a lot of time.

Chương 8

Comparing results from different control algorithms in simulation.

At this time, three different controllers have been developed and added into the framework of ArduPilot with the controller-integrating module above: the original controller, new PID controller and Integral Backsteping controller. This chapter will focus on comparing the results of each control algorithm using SITL. The result of these tests will give a general idea about the advantages and disadvantages of each controller. It will also a very important step to check the behavior of each controller in different situation before flying with the real system.

8.1 Flight plans

Figure 8-1 and **Figure 8-2** introduce the flight plans used to test the new control algorithms with Software in the Loop simulation. **Table 8-1** and **table 8-2** right

below each figure indicates some details of each flight path such as the distance between waypoints, its altitude, etc.

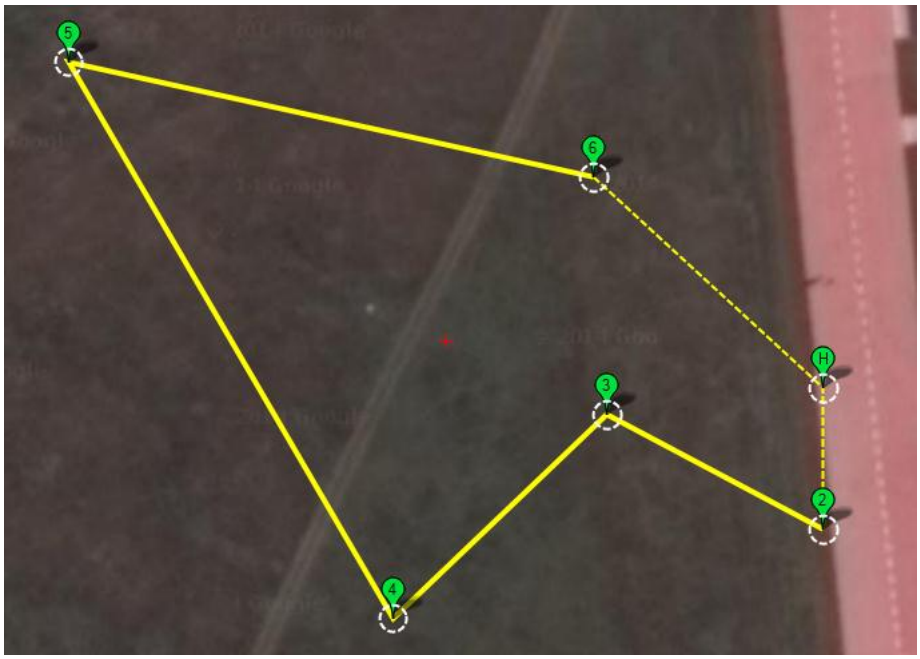


Figure 8-1: Flight plan 1

Table 8-1: Details of flight plan 1

Command	Delay [1] (second)	Latitude	Longitude	Altitude [2] (meter)	Distance from the previous waypoint (meter)
Takeoff	0	-35.363125	149.165085	10	0
Waypoint	0	-35.363125	149.165085	5	180
Waypoint	0	-35.362972	149.16473	3	298
Waypoint	0	-35.363246	149.164381	8	226
Waypoint	0	-35.362502	149.16385	20	330
Land	0	-35.362655	149.164708	0	102

[1] Time (in second) the UAV will wait at this waypoint before continue with the next one.
[2] Relative altitude from the absolute altitude of Home

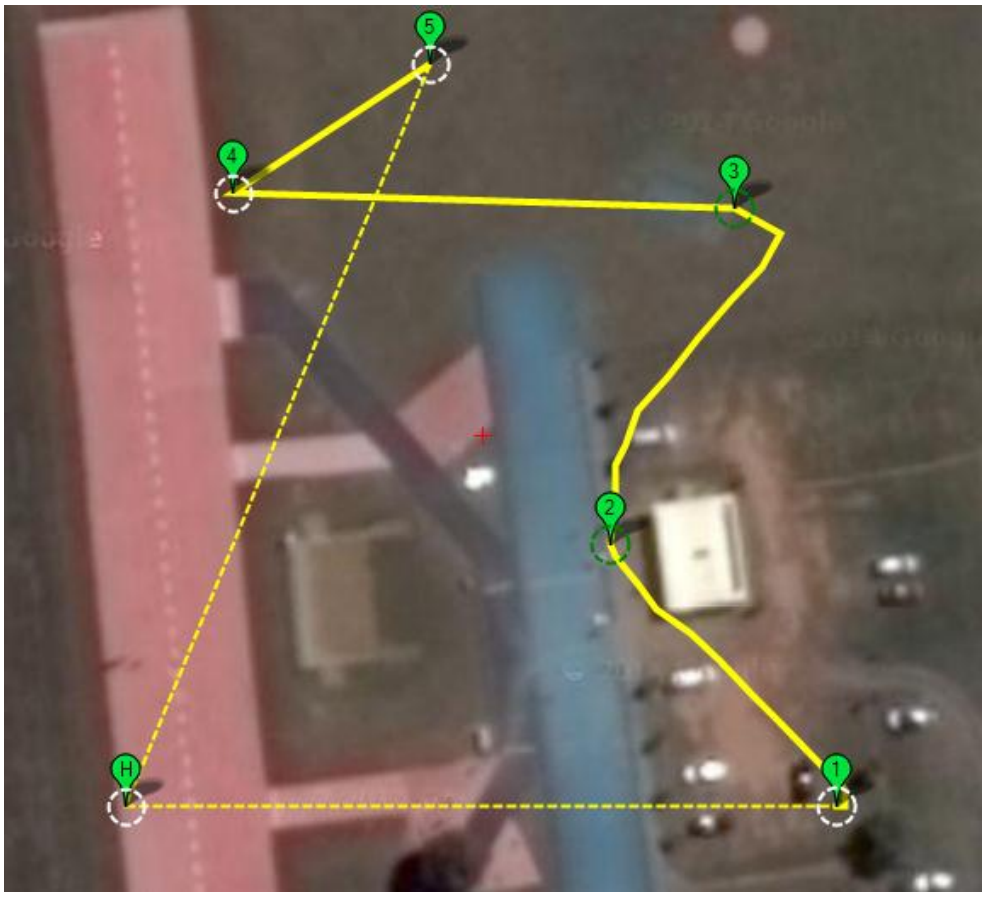


Figure 8-2: Flight plan 2

Table 8-2: Details of flight plan 2

Command	Delay (second)	Latitude	Longitude	Altitude (meter)	Distance from the previous waypoint (meter)
Waypoint	0	-35.362938	149.165955	5	78.9
Spline_wp[1]	0	-35.362675	149.16568	8	38.4
Spline_wp	10	-35.362339	149.165833	20	39.9
Waypoint	10	-35.362323	149.165215	10	56.1
Land	0	-35.362194	149.165457	0	26.2

As can be seen, with each mission, the new control algorithm will face several types of command. More importantly, in each case, not only the ability to maintain the stability but also the ability of tracking in various situations will also be tested. Some disturbances is also added to see if the controller is robust.

[1] A special waypoint which is used to define a spline flight path as can be seen in the figure.

Figure 8-3 presents the tracking result in simulation with the first mission using the original controller of ArduPilot. More information about the environment and the preflight settings of virtual sensors can be found in **Appendix F**.

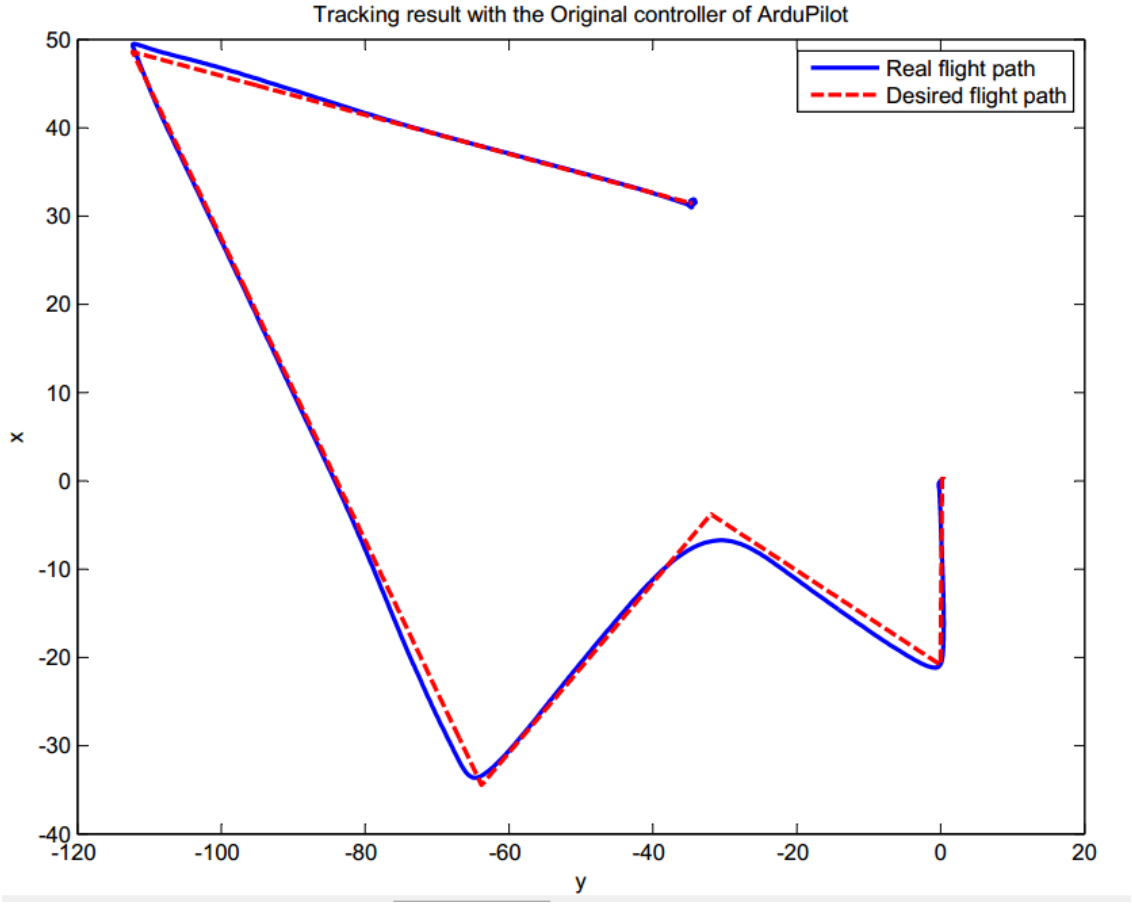


Figure 8-3: Mission 1 tracking result with original PID controller

8.2 New PID Controller

The PID gains for this test can be seen in **table 8-3** below, the values with the uppercase name is the name of the control constant related to the original PID controller. By using these variables, developers can change the PID gains of the new controller by using the ground control station and MAVLink protocol. More detail about this change can be found in **Appendix C**.

Table 8-3: PID gains

Altitude PID controller		
Gains	**Setting value**	**Default value**
P	THROTTLE_ACCEL_P*3.0	1.5
I	THROTTLE_ACCEL_I/10.0	0.1

D	THROTTLE_ACCEL_D	0.0
Phi (Roll) PID controller		
Gains	**Setting value**	**Default value**
P	RATE_ROLL_P	0.15
I	RATE_ROLL_I	0.1
D	RATE_ROLL_D + 2.0	2.004
Theta (Pitch) PID controller		
Gains	**Setting value**	**Default value**
P	RATE_PITCH_P/5.0	0.015
I	RATE_PITCH_I	0.1
D	RATE_PITCH_D + 1.0	1.004
Psi (Yaw) PID controller		
Gains	**Setting value**	**Default value**
P	RATE_YAW_P	0.2
I	RATE_YAW_I	0.02
D	RATE_YAW_D +2.0	2.0

As can be seen in **figure 8-4**, the simulation result of this new controller is good. The stability of the system, as can be seen in the HUD, is also acceptable

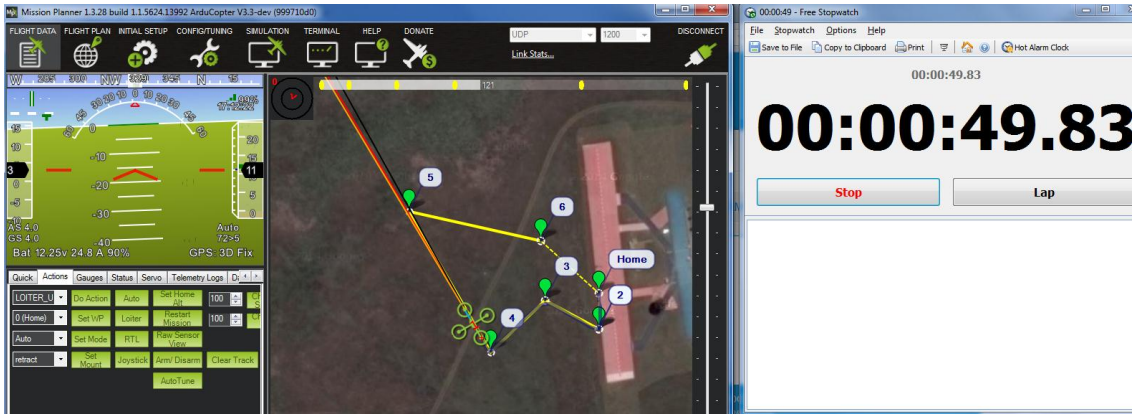


Figure 8-4: Testing with the first flight plan

At the place where there is a sudden change in the moving direction such as at the end of each waypoint, there are some errors between the real flight path and the desired one as can be seen in **figure 8-5**. However, these errors also happen in case of the original controller (**figure 8-3**) and it can be seen that new PID controller handles these error as good as the original does, in some case even better.

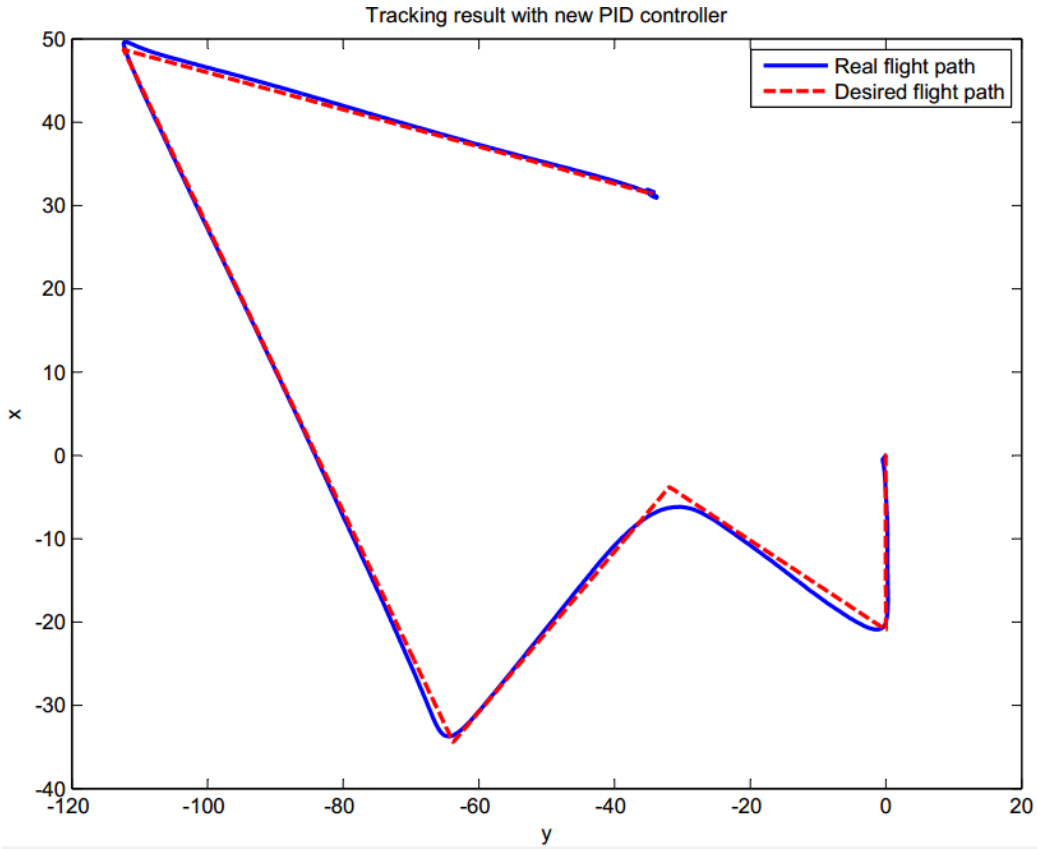


Figure 8-5: Mission 1 tracking result with new PID controller

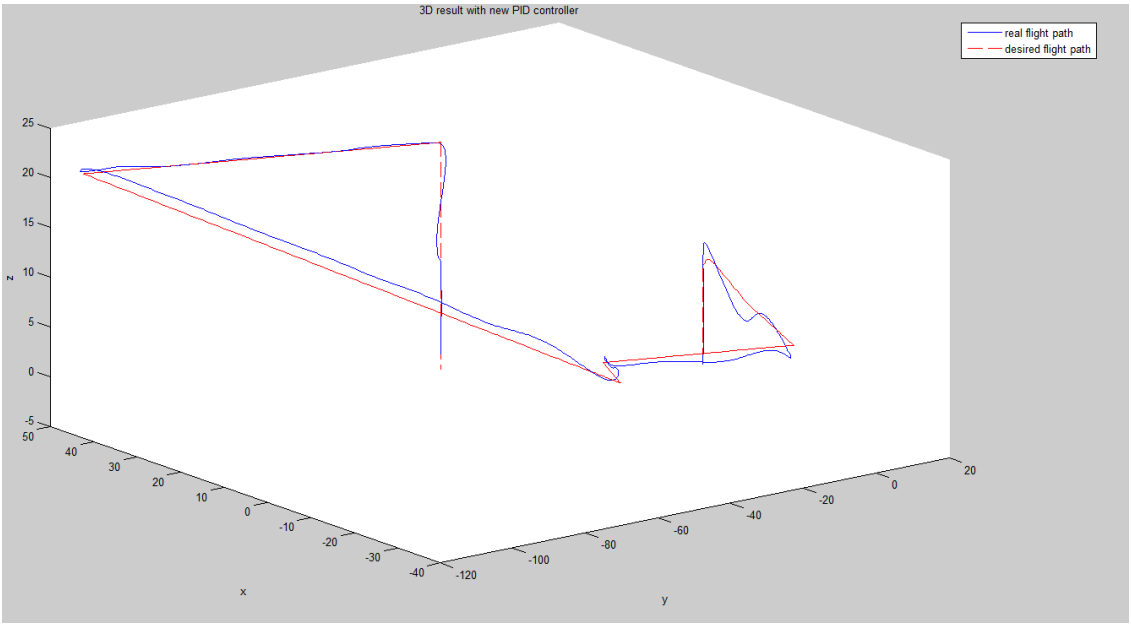


Figure 8-6: Mission 1 3D result with new PID controller

8.3 Simple Integral Backstepping Controller

The gains for this controller can be found in **Appendix F**. **Figure 8-7** shows a simulation results with the first flight plan with the simple IB controller. As can be seen from that figure, the tracking ability of this controller is even better than the two PID controllers above.

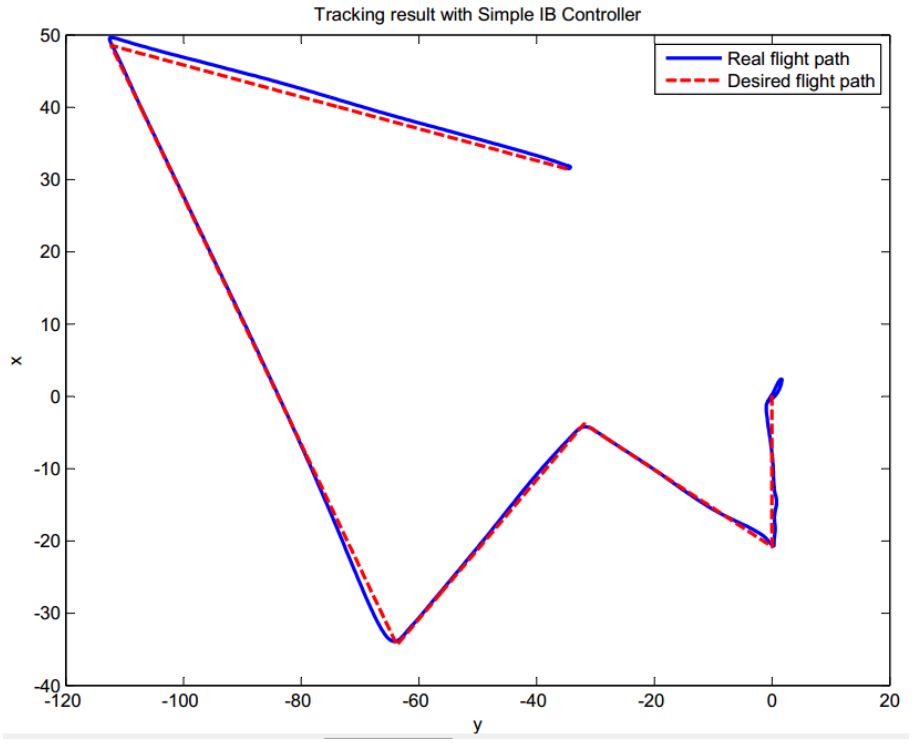


Figure 8-7: Simulation result with Simple Integral Backstepping Controller

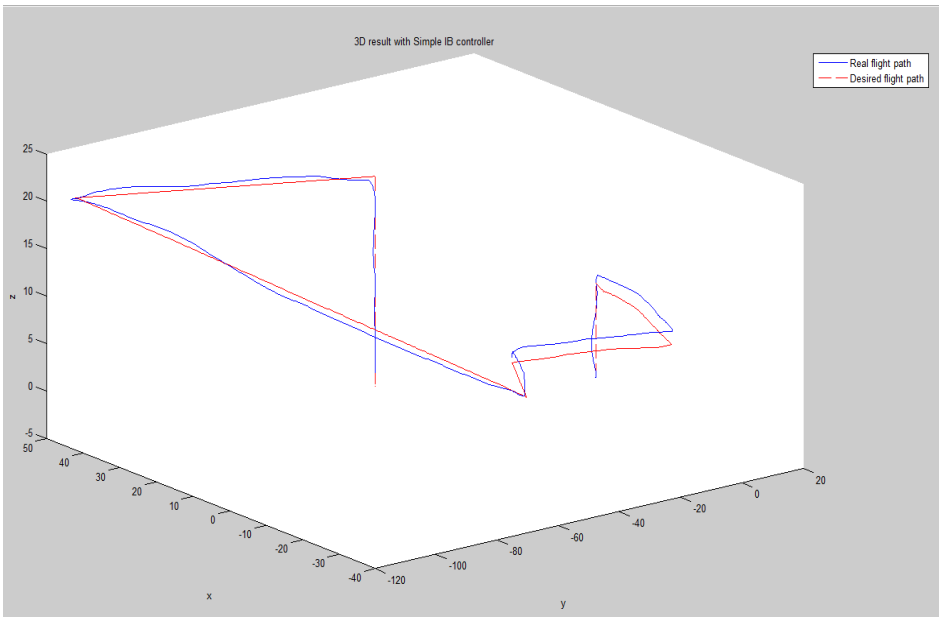


Figure 8-8: Mission 1 3D result with Simple IB controller

8.4 Comparisons

To compare the controllers, two separated groups of results will be considered. The first one related to the time finishing each mission and the amount of the battery left according to the simulation. The latter will compare the differences between the desired values (positions and angles) generated by the trajectory generator of ArduPilot and the response of the systems. Using the analyzing tool provided by Mission Planner, MATLAB data files are created for each time testing with the simulation.

Table 8-4 presents the results related to the first group.

Table 8-4: Results of each control algorithm

Flight plan	Control algorithm	Time to finish the tasks (minute : second)	Remained battery
Mission 1	Original PID controller	2:01.38	75%
	New PID controller	2:03.83	72%
	Simple IB Controller	2:30.40	70%
Mission 2	Original PID controller	1:54.27	77%
	New PID controller	2:00.80	74%
	Simple IB Controller	1:59.69	76%

From the table above, it can be seen that the results of the new PID controller are very close to those of the original one. The simple IB controller results are also acceptable, in the simulation with the spline waypoints, its results even better than those of the new PID controller.

Using the data from the MATLAB files, we can have a more detailed evaluation.

8.4.1 New PID controller and original PID controller

Figure 8-9 introduces the differences between desired pitch (generated by the differences between current position and position of position target) and the real pitch generated by the response of the quadricopter in simulation with the first flight plan above.

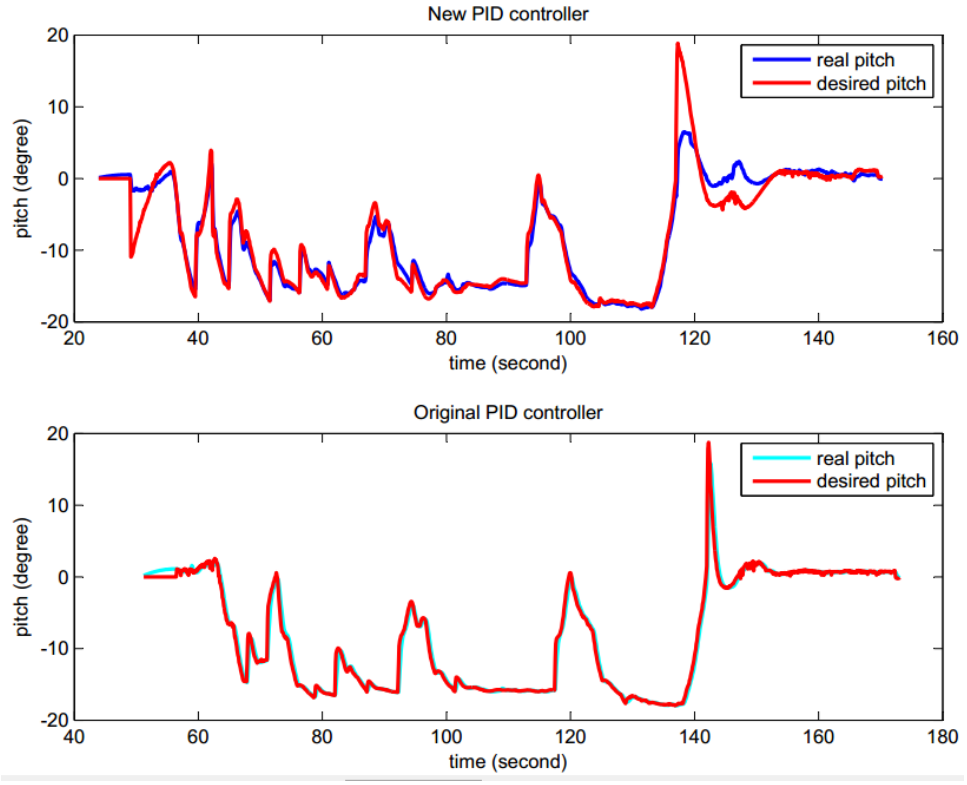


Figure 8-9: Difference between desired pitch and real pitch of the new PID controller and the original PID controller

The upper graph belongs to the values with the test using new PID controller algorithm, meanwhile, the lower indicates these values of the original controller. In both graphs, the horizontal axis is the time in seconds and the vertical one represents degree value. As can be seen in the upper graph, although there are differences between the desired pitch (red line) and the result pitch (blue line), both of them share a familiar tendency, going up and down together. The differences in the lower are less, the desired pitch and the real pitch seems to be one. There are two remarkable points with this figure. Firstly, the original controller of ArduPilot is very impressive and it is clear that this controller is nearly optimized. However, the new control algorithm is also very good, although there is overshoots due to the drawbacks as mentioned above, this controller can not only maintain the stability of the system but also works well with the tracking system. Since the gains of this controller still can be adjusted, the result still can be improved.

Figure 8-10 and **figure 8-11** introduce the results of roll and yaw from the same tests respectively. **Figure 8-12** shows the differences between the real altitude and the desired altitude.

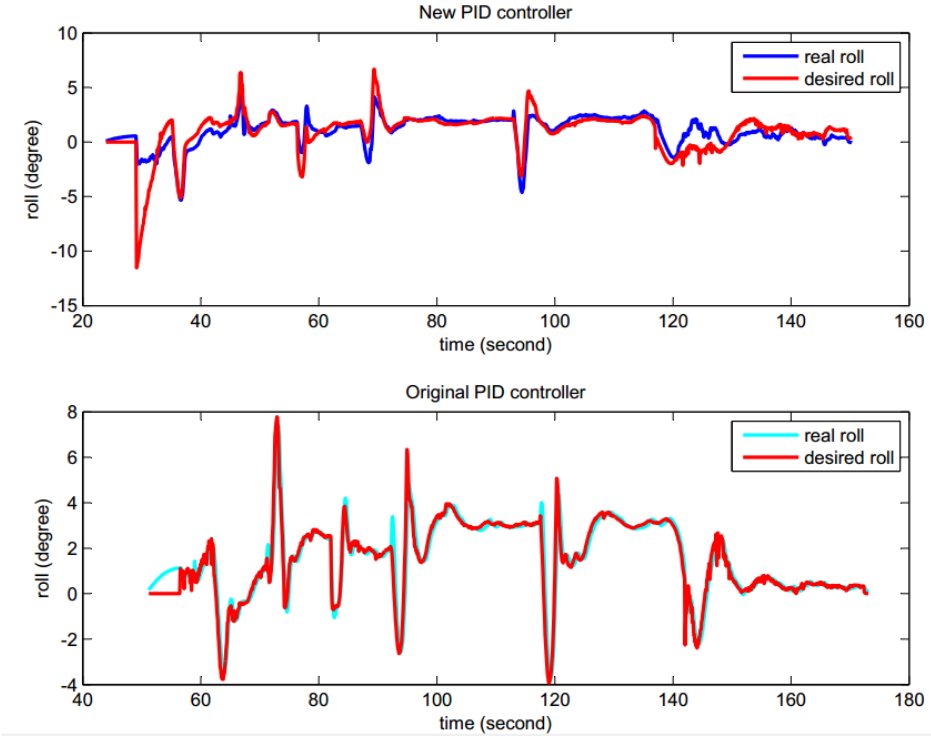


Figure 8-10: Roll and desired roll with new and original PID controller

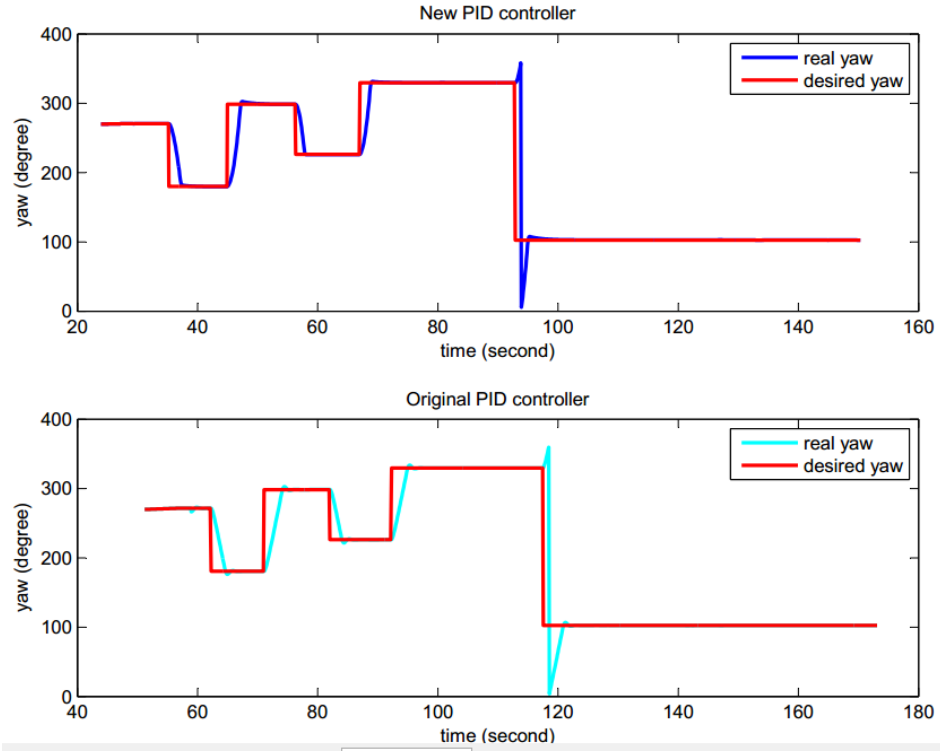


Figure 8-11: Yaw and desired yaw with new and original PID controller

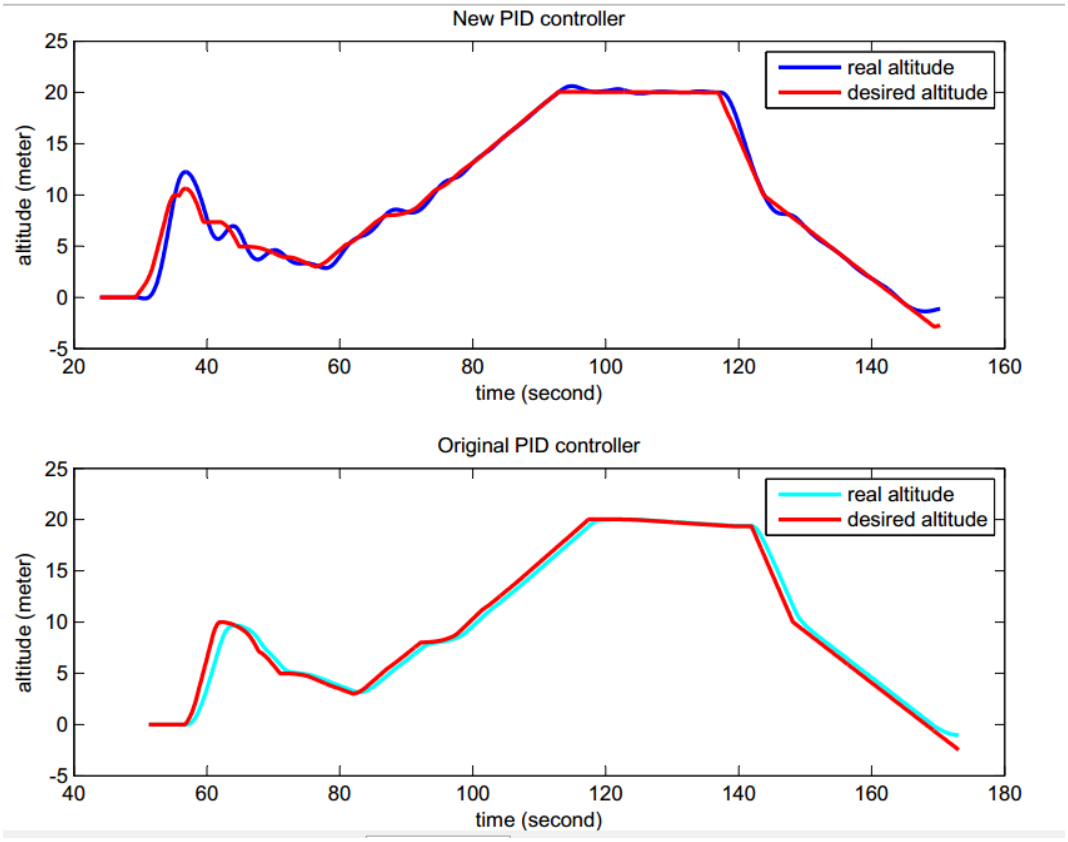


Figure 8-12: Real relative altitude and desired altitude of the simulation using new PID controller and original PID controller.

8.4.2 Simple Backstepping controller

Figure 8-13 presents the pitch results in simulation using the simple IB controller (upper) and the original controller of ArduPilot (lower). Even though the result of the new controller is not as good as the old one, simple IB controller seems to be as good as the new PID controller above. Although the result of this controller seems to be more unstable than that of the new PID controller, the differences between the desired values and the real values of the simple IB controller are much smaller, as can be seen clearly in **figure 8-14**. In other words, the response of this controller is better than the response of the new PID controller.

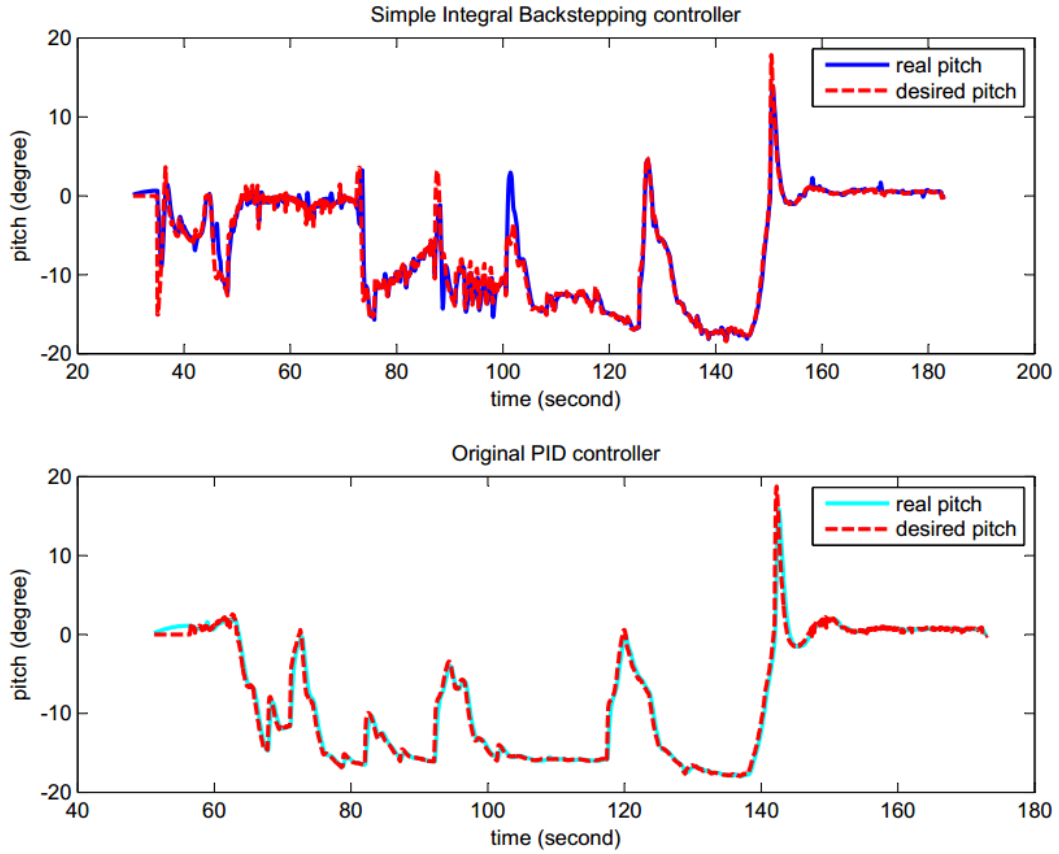


Figure 8-13: Difference between desired pitch and real pitch of the simple IB controller and the original PID controller

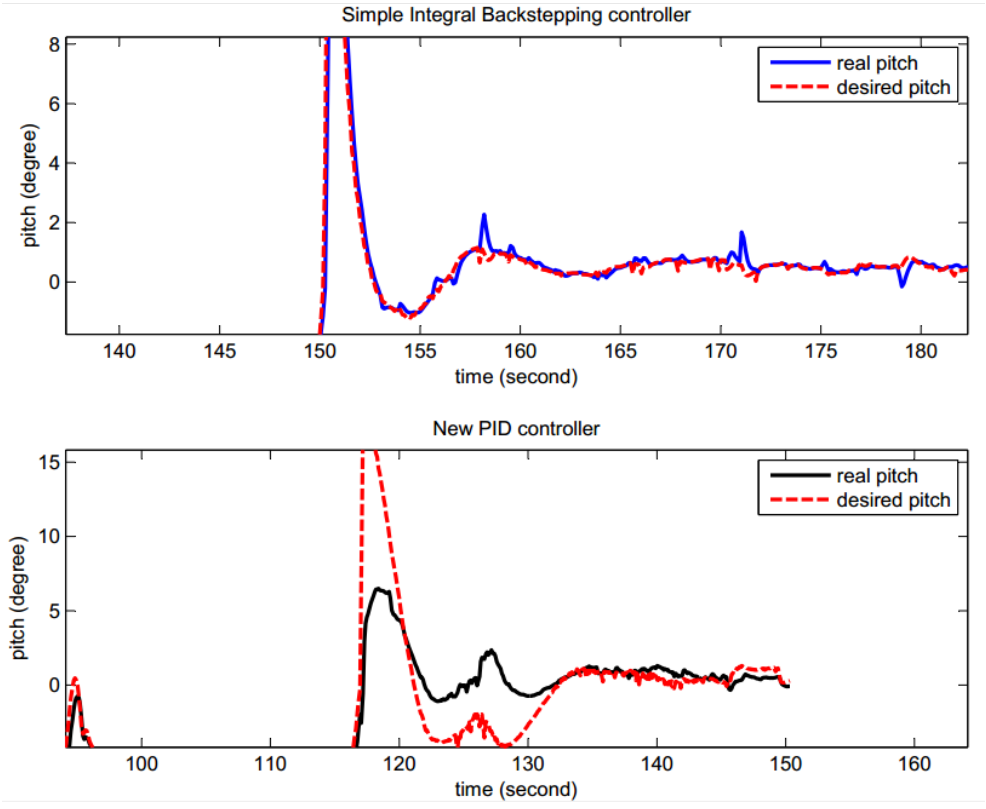


Figure 8-14: Differences between Simple IB controller and new PID controller

More comparisons and results related to the simple Integral Backstepping controller can be found in **Appendix G**.

8.5 Conclusion

It can be concluded that even though each controller has its own advantages as well as disadvantages when compared with the others, the original PID controller of ArduPilot seems to be the best controller among the three. Nevertheless, in some cases, the Simple IB controller has proved that it can handle the stability as well as tracking better than the original one. This result is suitable with the results from other researches, as can be seen in [**33**], which indicates that Integral Backstepping control algorithm is one of the most optimized controllers to control the quadricopter with an autopilot. The results of the full IB controller can be even better than the results of the Simple IB controller above.

When the number of controller increases, more conclusions about these controllers can be made. Based on these results, the suitable controller for specific requirements can be decided. For example, in case of the quadricopter working in an open area, the priority of the robustness of the controller must be higher than that in case flying in a small area, where the ability to avoid the obstacles is more important. These results also shows that the ArduPilot with just some modifications can become the framework that suitable for many different controllers and developers can create and integrate new controllers for specific needs.

Chương 9

Results with the real quadricopter

This chapter will presented the results obtained so far with the real model. Moreover, in order to help the future works, it will also present other configurations with the real model. This section is divided into the two subsections:

- *Hardware preparations*
- *Sensors calibration and safety procedure*

9.1 Hardware preparations

9.1.1 Setting up connections and position for components

9.1.1.1 Embedded system and electric system

As mentioned in [**16**], FlyMaple can connect directly with the Wi-Fi shield, the GPS shield and then use the Vin Pin of FlyMaple to supply the power for the whole system. However, while using the battery and the power from the ESC, that solution will not work. In this case, in order to optimize the size and the weight of the embedded system, the powering would be as shown in **table 9-1**.

Table 9-1: Powering the embedded system

Board	Vin	Powering Pin	Receiving power from
FlyMaple	5V	Middle leg of any Pin connecting with the ESCs (D28, D27, D11, D12) (one red wire only)	ESC red wire
Wi-Fi shield	5V	+5V	Pin 5V of FlyMaple
GPS shield	5V	Vin	Pin 5V of Wi-Fi shield

From the table above, there must be a jumping wire connecting Pin 5V of Wi-Fi shield and Vin of GPS shield. **Figure 9-1** and **figure 9-2** gives the overview of the completed embedded system.

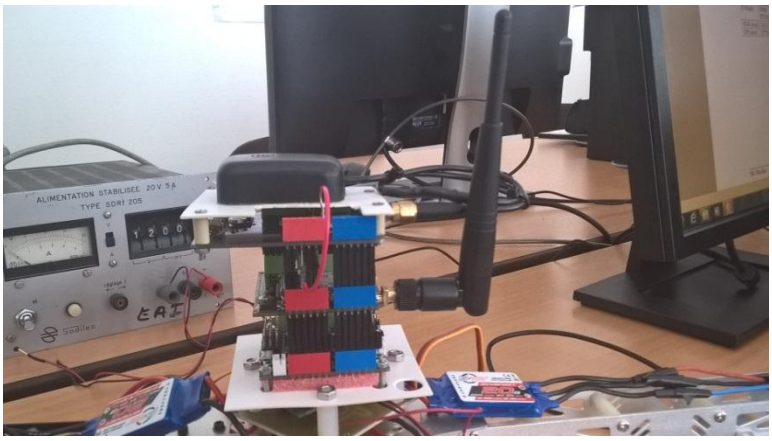


Figure 9-1: Completed control system

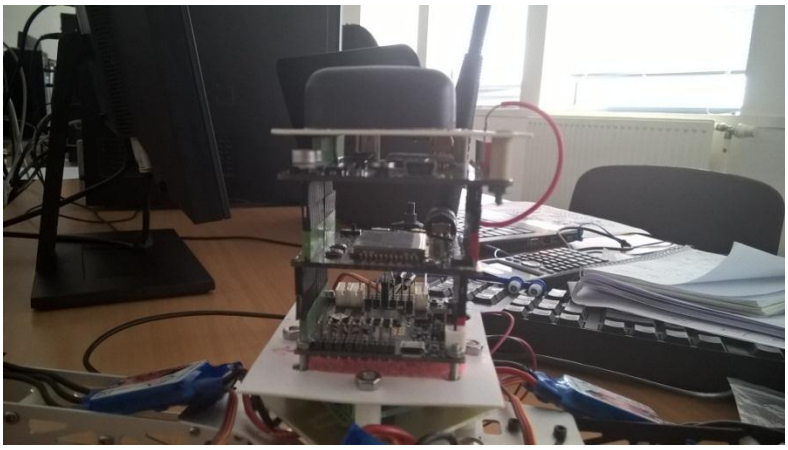


Figure 9-2: Completed control system (2)

9.1.1.2 Completed frame

With the rotation of four rotors at very high angular speed, the frame will be vibrated. In order to reduce the vibration and its effect on the sensors, additional

damper system have to be constructed. There are several ways to do that, however, using the available material, a simple vibration damper has been created as can be seen in **figure 9-2.** More solutions for this problem can be found in [**34**].

Figure 9-3 present the complete real quadricopter without the propellers. The weight of the whole system is now 1.14 kg. Some modifications need to be done in the movement equations to get the right values with this new weight. Other characteristics of this quadricopter and its components can be found in **Appendix A**.

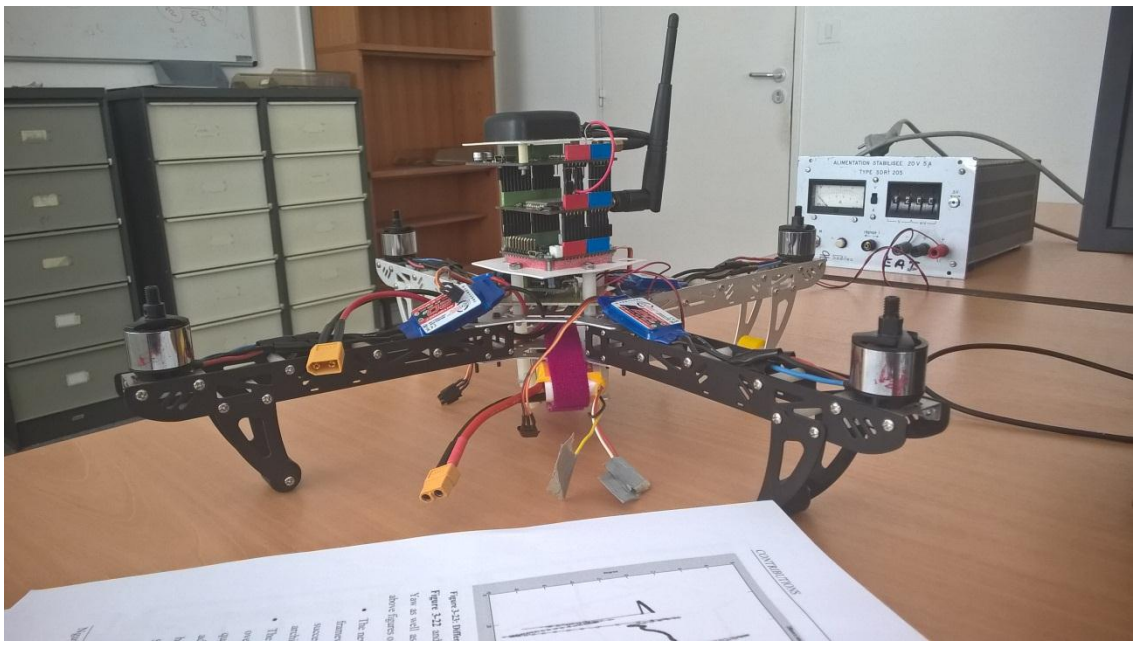


Figure 9-3: Completed quadricopter without propellers

9.1.2 GPS shield configurations

By defaults, even though GPS shield can communicate with FlyMaple as discussed in [**16**], the message from this module cannot be understood by a firmware compiled with ArduPilot and vice sera. Some configurations have to be done to make these system able to communicate.

With a GPS shield using ublox GPS chipset as the one in this project, tests to check the shield as well as the configuration to make it compatible with ArduPilot can be done with **U-center GNSS evaluation software**. More detail

about this tools, the procedure to install it and the process to establish a connection between this tools and the shield can be found in [**35**], [**36**] and [**37**]. **Figure 9-4** introduces a working GPS system in U-center and **Figure 9-5** indicates a working GPS shield on a firmware built with ArduPilot.

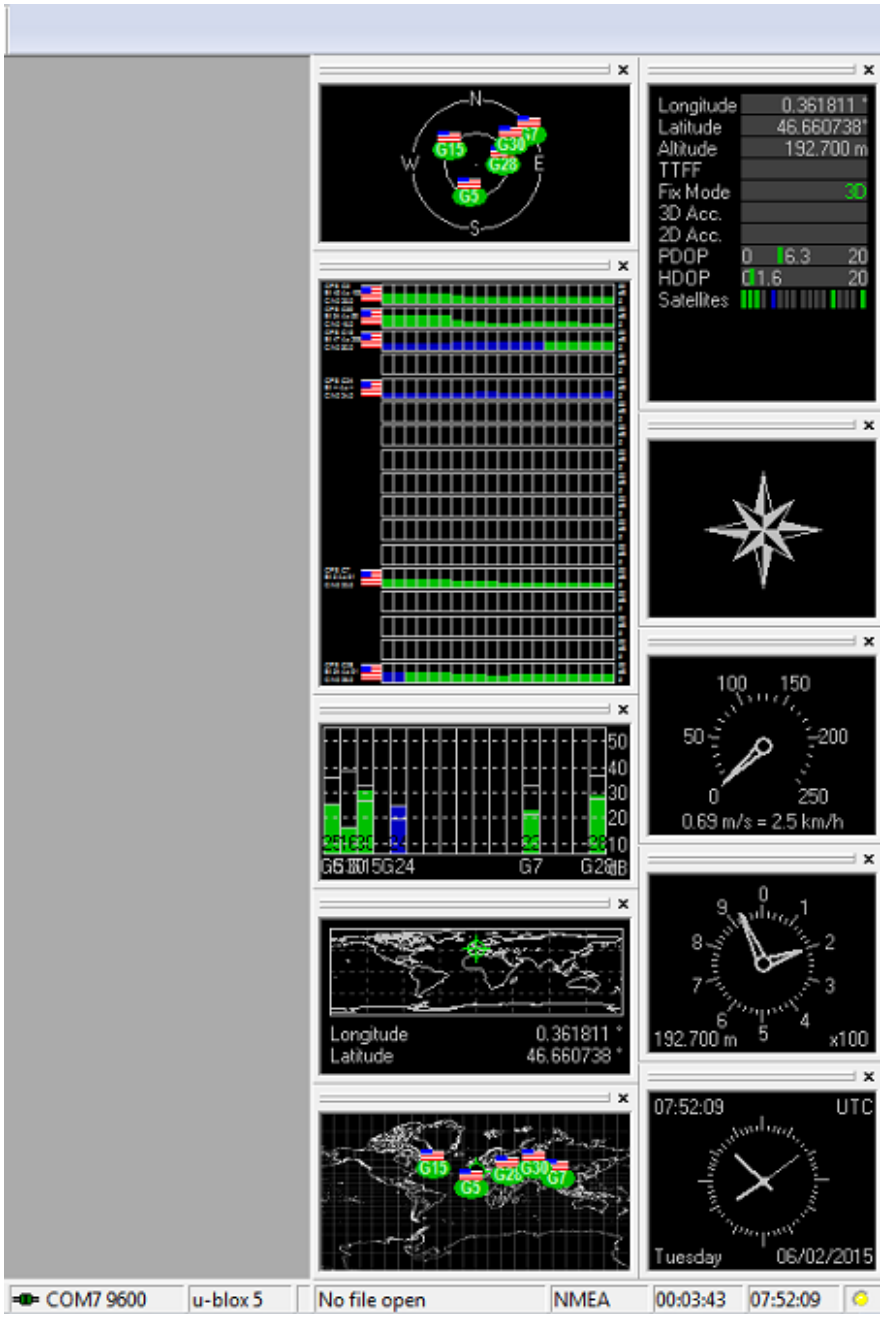


Figure 9-4: A working GPS shield with U-center

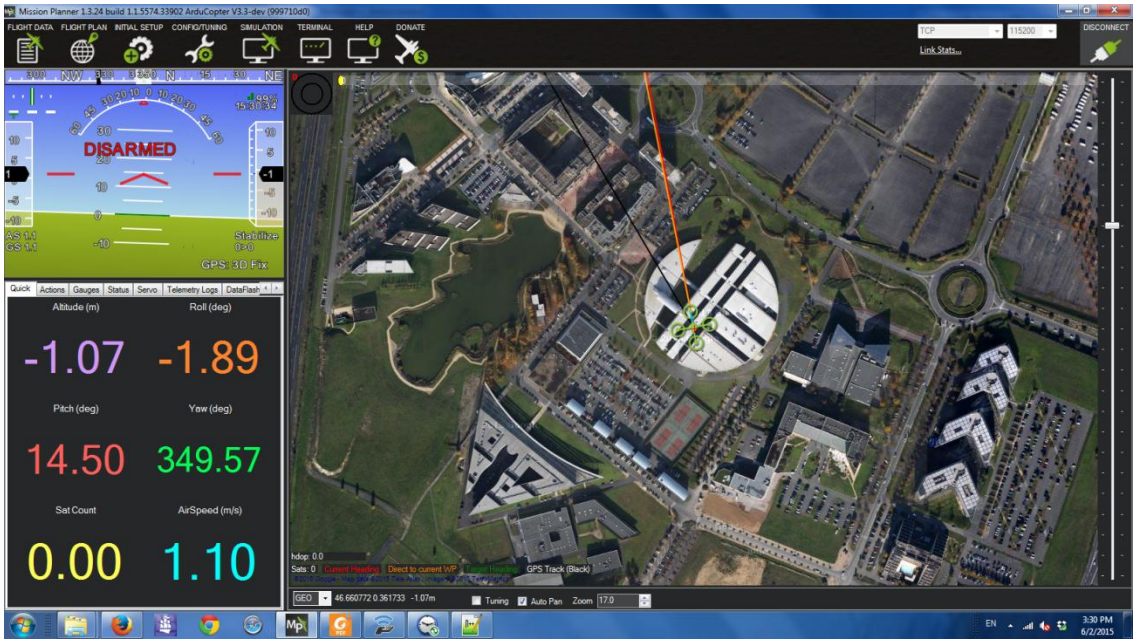


Figure 9-5: GPS fixed and location display with Mission Planner

9.1.3 ESC calibration using ArduPilot

In preflight configuration, there are two specific tasks related to the ESC: calibration and programming. The objective of these tasks is for the rotors to have the same angular speed and to rotate correctly when receiving the same PWM signal.

- Calibration

In short, Electric Speed Control calibration is a procedure used to teach the ESC the value of the biggest and the lowest PWM signal. After calibrated, ESC will control the rotor at full throttle (100%) when receiving the maximum and then zero throttle (0%) with minimum value. The PWM signal sent to an ESC must be between 800µs and 2200µs, which means 800 and 2200 for the min and the max respectively. These values are suitable for almost every RC controller and receivers.

A common ESC will require the control signal to have a frequency varying from 50 to 150 Hz. With some ESC, this value can reach 400 Hz. Any signal with a lower frequency will not be accepted, meanwhile, a signal with faster rate will be neglected. Knowing the signal sending frequency of each control board/firmware

will be essential to calibrate the ESC correctly since this value will affect the sending values from the control board. In fact, in case of controlling the ESC via the control board, there are two groups of PWM values. The first group, which is the nominal values, is the one that the users define, which is between 900 and 2000µs. The latter, which is the signal sent from the control board, is a function of the nominal value, the operating frequency, the timer of the control board, etc.

Although the ESC is usually calibrated using a RC controller, by understanding the fact that this process is nothing but sending the maximum and then minimum PWM value, a code can be generated to calibrate the ESCs. Moreover, to make sure that the calibrated ESC operates properly with the firmware built by ArduPilot, a simple specific firmware just for calibrating ESCs is created using ArduPilot. The procedure to calibrate an ESC with the developed firmware can be found in **Appendix F**.

- Programming

Although an ESC can be used at manufacturer settings and these settings can still be used to control a quadricopter, some modifications can be done to optimize the operation of each rotor. The most important change that has to be done is setting the rotation direction of each motor. As mentioned above, in quadricopter, there are two propellers rotating clockwise and the other two counter-clockwise. A simple changing with the wiring can be done to reverse the rotating direction of a DC brushless motor … However, in case the connections between ESC and rotor cannot be changed, users can still change the rotation direction of a motor manually by programming the ESC. In fact, the code used to program the ESC is the one used to calibrate the ESC, and this project has used the specific firmware mentioned above to program the ESCs. More information about this can be found in [**38**] and [**39**]. The final setting of the ESC in this project can be seen in **table 9-2**.

Table 9-2: ESC settings

Configuration	Setting
Minimum PWM value	900
Maximum PWM value	2000
Brake	Brake off
Battery type	Li-Po
Low Voltage Cut-off Threshold	High (3.2V/65%)
Timing Setup	Automatic
Switching Frequency	8kHz

9.2 Pre-flight settings

9.2.1 Understanding the safety procedures of ArduPilot

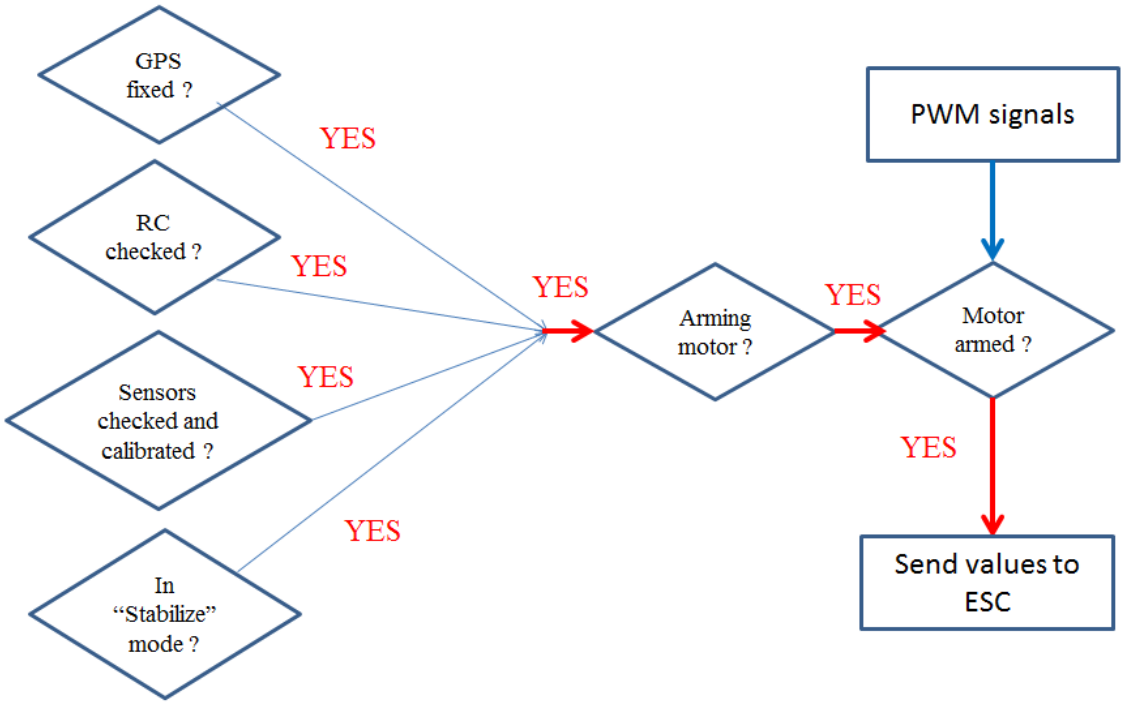


Figure 9-6: Requirements for arming command

Figure 9-6 presents the requirements needed to be done before an arming command can be accepted. When the motors are not armed, there will be no signal sent from the control board to the ESC. As can be seen from the figure, there are various checks before arming the motors.

With the sensors, one of the most important tasks before a flight is calibration, particularly for the compass and the accelerometer. For the first one, since the magnetic field is not the same for every place in the earth, compass calibration

process will tell the compass which direction is the North, make the navigation system more precise. Moreover, the working of the compass will be affected by the surrounding environment, mostly by the metals of the frame and the magnetic field created by the wires. The calibration process will add this effect into calculation, increase the stability and the precision of the control. ArduPilot also has a limit for the offset values after calibration, which will check if the calibration process has been done correctly or not. In this project, an internal compass is used. With the internal compass, since the metal of the board will affect the magnetic field around the compass, the final offset values are bigger than the safety value. As mentioned in [**40**], in this case, the compass check can be disabled.

With the accelerometer, the calibration procedure will get the difference between the earth frame and the body frame. This will allow it to understand which values mean the system is stable, which is essential for stability and autopilot. However, since the accelerometer in this project cannot be calibrated neither by the tools of ArduPilot nor by manually generated code, and because of the response of the quadricopter with the HUD of the GCS is good as expected, this step will be skipped.

In order to skip these pre-arm checks, the value of the parameter **ARMING_CHECK** will be changed as shown in **figure 9-7**. After this change, the motors can be armed if the flight mode is "stabilize".

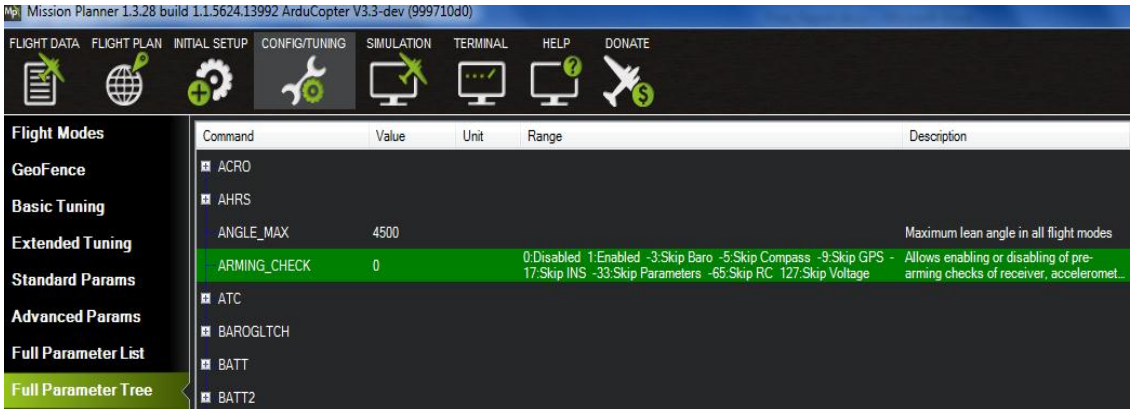


Figure 9-7: Disabling all the pre-arming check

9.3 .Real flight results

At the end of this project, the quadricopter is still in testing for the stability in assisted mode and for the gains of the original PID controller. More testing need to be done before the real things can be tested in auto mode.

Chương 10

Conclusions and perspectives

10.1 Achievements

According to the author, these are the achievements of this project:

- A solution to use the firmware built with ArduPilot framework for a FlyMaple control board has been presented.
- A Ground Control Station has been chosen based on the tests with the real control board and the ArduPilot firmware.
- A new module to embed new control algorithms into the structure of ArduPilot has been established successfully. Until now there are three different controllers that have been integrated into ArduPilot and users can choose the controller with just a single change.
- The results of the new controllers are presented and compared with the the original controller of ArduPilot.
- Preflight settings and configurations for the real quadricopter have been done and ready for the real test.

10.2 Limitations and next steps

There are various limitations with this project:

- The quadricopter has not been tested with the real flight test.
- The IB controller, which should be the best controller based on the reference, has not been embedded successfully.
- The number of the control algorithm is still small, which leads to the lack of results to compare and conclude.
- The compatibility of the new controllers is still limited, making the new controllers cannot use all of the facilities of the framework.

With the achievements and the limitations above, the objectives for the future works would be:

- Test the real system
- Finish the work with the IB controller and test the full IB controller with the simulation and the real flight test
- Add more control algorithms and continue optimizing the controller for the quadricopter.
- Ease the step of transformation from Simulink to ArduPilot.

Tài liệu tham khảo - References

[1] US Department of Defense, *Dictionary of Military and Associated Terms*., 2005. [Online]. http://www.thefreedictionary.com/Unmanned+Aerial+Vehicle

[2] H.McDaid et al. (2013, Feb) Remote Piloted Aerial Vehicles : An Anthology. [Online]. http://www.ctie.monash.edu/hargrave/rpav_home.html#Beginnings

[3] J.F Kean and S.S.Carr, "A brief History of Early Unmanned Aircraft ," *Johns Hopkins APL Technical Digest*, vol. 32, pp. 558-571, 2013.

[4] (2005, July) The Radioplane Target Drone. [Online]. http://www.ctie.monash.edu.au/hargrave/rpav_radioplane4.html

[5] Â Frost and Sullivan, "Study Analysing The Current Activities in The Field of UAV," European CommisionÂ Enterprise and Industry Directorate-General, Technical Report 2007. [Online]. https://engineering.purdue.edu/HSL/index.php?page=numerical-analysis-of-cyberattacks

[6] THE NEW YORK TIMES. (2010) Elmer Sperry Dies; Famous Inventor (Jun 1930). [Online]. http://www.nytimes.com/learning/general/onthisday/bday/1012.html

[7] William Scheck. (2006, Dec) Lawrence Sperry: Autopilot Inventor and Aviation Innovator. [Online]. http://www.historynet.com/lawrence-

sperry-autopilot-inventor-and-aviation-innovator.htm

[8] "Now - The Automatic Pilot," *Popular Science Monthly*, p. 22, Feb 1930. [Online]. http://books.google.fr/books?id=4ykDAAAAMBAJ&pg=PA22&redir_esc=y#v=onepage&q&f=false

[9] William Scheck. Development of Aviation Technology. [Online]. http://www.century-of-flight.net/Aviation%20history/evolution%20of%20technology/autopilot.htm

[10] Federal Aviation Administration, "Automated Flight Control," in *Advances Avionics handbook*., 2014, ch. 4, pp. 4-2.

[11] M.Orsag et al, "Hybrid Fly-by-Wire Quadrotor Controller," ISSN 0005-1144, 2010.

[12] Samir Bouabdallah, "Design and Control of quadrotors with application to autonomous flying," École Polytechnique Fédérale de Lausanne, PhD Thesis 2007.

[13] Connecting the ESCs and Motors (APM2). [Online]. http://copter.ardupilot.com/wiki/connecting-the-escs-and-motors/

[14] A.Castillo Benito, "Flight Control and Navigation of a Quadcopter," ISAE-ENSMA, Poitiers, Internship Report 2014.

[15] (2015) The Tech Terms Computer Dictionary. [Online]. http://techterms.com/definition/firmware

[16] N.Đạt BÙI, "Embedded System for Quadricopter," Ho Chi Minh City University of Technology, Internship Report 2014.

[17] (2015) Welcome to the ArduPilot/APM development site. [Online]. http://dev.ardupilot.com/

[18] (2015) Learning the ArduPilot Codebase. [Online]. http://dev.ardupilot.com/wiki/learning-the-ardupilot-codebase/

[19] Daniel Petri. (2009, Aug) OSI Model Concepts. [Online]. http://www.petri.com/osi_concepts.htm

[20] A.D. Marshall, "New Feature of C++," in *Hands on: C++ Programming*., 1998-2004, p. 24.

[21] (2015) Learning ArduPilot - Threading. [Online]. http://dev.ardupilot.com/wiki/learning-the-ardupilot-codebase/learning-ardupilot-threading/

[22] Building ArduPilot for FlyMaple in Linux. [Online]. http://dev.ardupilot.com/wiki/building-the-code/building-apm-for-flymaple/

[23] Open-Source MAVLink Micro Air Vehicle Communication Protocol. [Online]. http://qgroundcontrol.org/mavlink/start

[24] MAVLink Onboard Integration Tutorial. [Online]. http://qgroundcontrol.org/dev/mavlink_onboard_integration_tutorial

[25] (2015, Mar) Cyclic redundancy check. [Online]. http://en.wikipedia.org/wiki/Cyclic_redundancy_check#Commonly_used_and_standardized_CRCs

[26] Waypoint Protocol. [Online]. http://qgroundcontrol.org/mavlink/waypoint_protocol

[27] Flight Modes. [Online]. http://copter.ardupilot.com/wiki/flying-arducopter/flight-modes/

[28] Martin Gomez. (2001, Nov) Hardware-in-the-Loop Simulation. [Online]. http://www.embedded.com/design/prototyping-and-development/4024865/Hardware-in-the-Loop-Simulation

[29] SITL Simulator (Software in the Loop). [Online]. http://dev.ardupilot.com/wiki/simulation-2/sitl-simulator-software-in-the-loop/

[30] Setting up SITL on Windows. [Online]. http://dev.ardupilot.com/wiki/simulation-2/sitl-simulator-software-in-the-loop/setting-up-sitl-on-windows/

[31] Setting up SITL on Linux. [Online]. http://dev.ardupilot.com/wiki/simulation-2/sitl-simulator-software-in-the-loop/setting-up-sitl-on-linux/

[32] S. John A. Zulu, "A Review of Control Algorithms for Autonomous Quadrotors," *Open Journal of Applied Sciences*, pp. 547-556, April 2014.

[33] Bouabdallah.S et al, "Full control of a quadrotor," in *Intelligent Robots and Systems 2007. IROS 2007*, San Diego, 2007, pp. 153-158.

[34] Vibration Damping. [Online]. http://copter.ardupilot.com/wiki/initial-setup/assembly-instructions/vibration-damping/

[35] u-center GNSS evaluation software for Windows. [Online]. http://www.u-blox.com/en/evaluation-tools-a-software/u-center/u-center.html

[36] UBlox GPS Configuration. [Online]. http://copter.ardupilot.com/wiki/common-optional-hardware/common-positioning-landing-page/common-installing-3dr-ublox-gps-compass-module/common-ublox-gps/

[37] (2015, Mar) U-center, GNSS evaluation software for Windows. User Guide.

[38] Motor Setup. [Online]. http://copter.ardupilot.com/wiki/initial-setup/motor-setup/

[39] molé3. ESC Programming on Arduino (Hobbyking ESC). [Online]. http://www.instructables.com/id/ESC-Programming-on-Arduino-Hobbyking-ESC/

[40] Pre-Arm Safety Check. [Online]. http://copter.ardupilot.com/wiki/flying-arducopter/prearm_safety_check/

[41] Daniel Barlow. (1996) What is ELF? An introduction. [Online]. http://cs.mipt.ru/docs/comp/eng/os/linux/howto/howto_english/elf/elf-howto-1.html

[42] Introduction to cross-compiling for Linux. [Online]. http://landley.net/writing/docs/cross-compiling.html#footnote1

[43] EmSys:An Introduction to the GNU Compiler. [Online]. http://shukra.cedt.iisc.ernet.in/edwiki/EmSys:An_Introduction_to_the_GNU_Compiler

[44] (2015) Building the code. [Online]. http://dev.ardupilot.com/wiki/building-the-code/

[45] (2015) Maple cant find dfu device. [Online]. https://www.olimex.com/forum/index.php?topic=465.0

[46] (2015) GNU make in detail for beginnners. [Online]. http://www.opensourceforu.com/2012/06/gnu-make-in-detail-for-beginners/

[47] R.McGrath, P.D.Smith R.M.Stallman. (2014, September) GNU Make.

[48] TCOM 370 Notes 99-9 Cyclic Codes and the CRC Code. [Online]. https://www.seas.upenn.edu/~kassam/tcom370/n99_9.pdf

[49] Field Reordering and CRC Extra Calculation. [Online]. http://qgroundcontrol.org/mavlink/crc_extra_calculation

[50] David Suarez. MavLink.NET. [Online].

https://github.com/dsuarezv/mavlink.net

[51] Lorenz Meier. MAVLink. [Online]. https://github.com/mavlink/mavlink

[52] QGroundControl - Home. [Online]. http://qgroundcontrol.org/start

[53] Mission Planner Overview. [Online]. http://planner.ardupilot.com/wiki/mission-planner-overview/

[54] APM Planner 2.0: Credits and Contributors. [Online]. http://planner2.ardupilot.com/home/credits-and-contributors/

[55] Using SITL for ArduPilot Testing. [Online]. http://dev.ardupilot.com/wiki/simulation-2/sitl-simulator-software-in-the-loop/using-sitl-for-ardupilot-testing/

[56] Paul Cantrell. Pauls at copter. [Online]. http://www.copters.com/aero/gyro.html

[57] Loiter Mode. [Online]. http://copter.ardupilot.com/wiki/flying-arducopter/flight-modes/loiter-mode/

Appendix

A.List of components and specifications

Table A-10-1: List of components of the quadricopter

Component	Notes
Frame	ST450
Control board	FlyMaple, a 10DOFs control board with internal sensors
Wi-Fi shield + Antenna	WizFi210
GPS shield + Antenna	DFRobot GPS shield with a ublox LEA-5H chip
Electric Speed Controller	RCplus Skysport 20 with a BEC 5V 2A
Rotor	HL2816A
Propeller	1038 and 1038P
Battery	Rhino Lithium Polymer 3S 2250mAh

Specifications of the completed model

Table A-1-2: Completed model specification

Parameter	Unit	Value
Total weight	g	1140
Three dimensional size	$mm \times mm \times mm$	$450 \times 450 \times 170$
Distance from center to rotor	mm	225

B.Cross-compilation and GNU make

Cross-compilation

The code from the codebase, or the code created by the users, which can be read and understood by human, is not the kind of language a hardware, in this case a controller board, can understand. In order to change from programming language to the machine language, a process called **"build"** or **"compile"** must be done.

In this case, it is a cross compiling, a process turning a source code (in this case C/C++ code) into executable code file, which can run in a different platform from the programming and the compile environment, called the host environment. For controller board, the output will be ***.elf** file, "*ELF (Executable and Linking Format) is a binary format originally developed by USL*" [**41**]. Cross-compiling is very important, in particularly for embedded systems, where the executive systems do not have enough resource to compile the running file itself, or programmers cannot write the particular code for that kind of language [**42**].

Step-by-step, the cross-compile from C/C++ source to executable file can be described in the figure below.

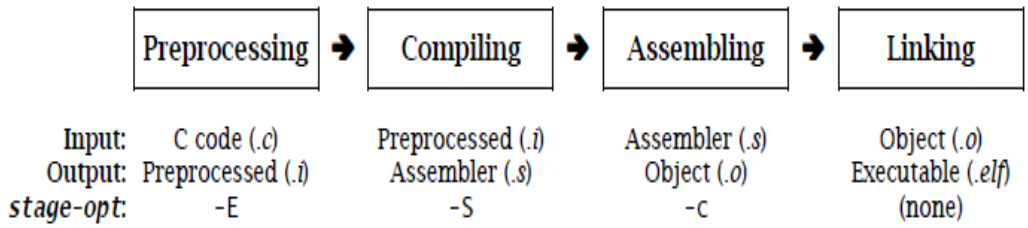


Figure B-1-1: Stages of Compilation [43]

More detail of cross compiling along with the file types of ArduPilot will be discussed further in the next section of this report.

The result of this process is called **firmware**. After being built, this firmware must be **"uploaded"** into the board.

There are several ways to build a firmware from ArduPilot platform. One of them is using an **I**ntegrated **D**evelopment **E**nvironment (IDE) such as **Eclipse** or **Emacs.** These programs not only provide the environment to build the firmware (the cross-compile tools), but also a working place to modify the code, find structure, etc. The modifications for these IDE to make them suitable for building ArduPilot code is in [**44**].

In general, making or building a firmware for any target needs three commands, as in **figure B-1-2**.

```
1  make clean
2  make flymaple-quad
3  make upload
```

Figure B-1-2: Commands to compile upload to main board in Terminal

The first command in **Figure B-1-2** will clean any unneeded old firmware as well as the intermediate files created by a previous compilation. Meanwhile the second one will build a specific firmware for **FlyMaple** board using to control the Quadricopter. After finish building the firmware, the last command will upload it into the board. **Figure B-1-3** below presents the process of uploading a built firmware indicated above. During uploading, instead of changing the board to perpetual boot loader mode every time as described in [**16**], we can simply quickly push the reset button at the right time after entering "make upload" command. During restart, the FlyMaple will have a few second in the boot loader mode and will receive the uploaded firmware [**45**].

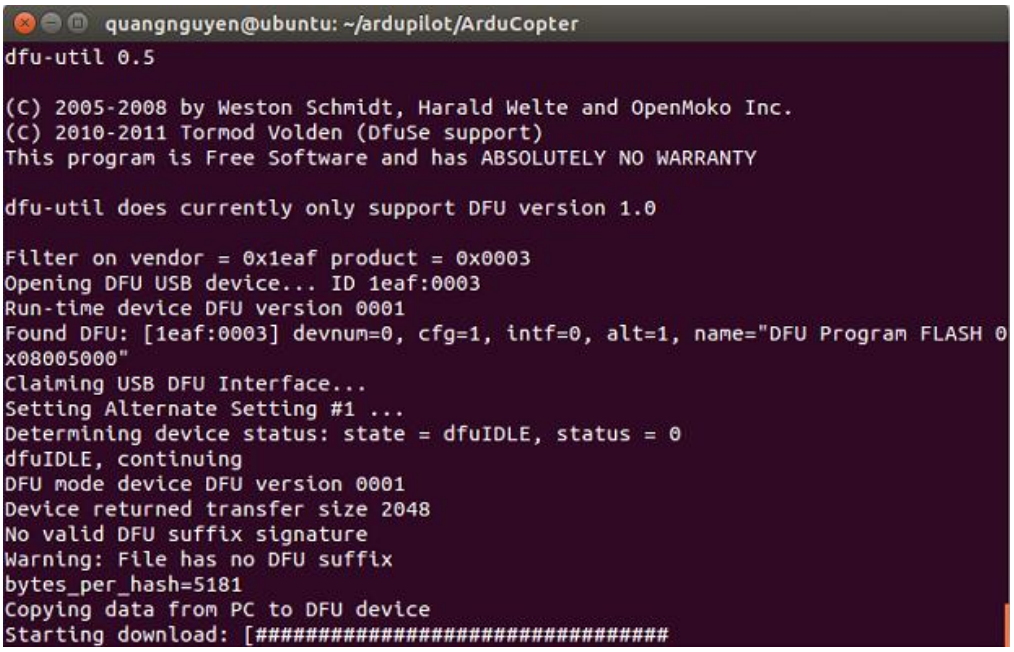


Figure B-1-3: Build successfully and uploading firmware

GNU make

As discussed above, compiling the firmware is using "make". Like any other open source program, which means thousands of code lines, many developers

and hundred source files, ArduPilot uses "make" as a solution to build the target program, in this case a firmware. As mentioned in [**46**], one important mission of *make* files is to define the dependency between source files, so that the compiling process with "make" will build in an order that satisfies these dependencies. This process is very important in case of a change and need to recompile, "make" can just recompile the changed file and any other parts that are related to it. This report will mostly focus on GNU make, which was implemented by Richard Stallman and Roland McGrath [**47**], since ArduPilot uses this particular make process.

The files structure of ArduPilot includes three different types of file. All of the files in the library, which define the abstract hardware layer, mapping hardware layer as well as particular functions, are ***.h** files and ***.cpp** files. Meanwhile, in ArduCopter folder, there are ***.h** and ***.pde** files.

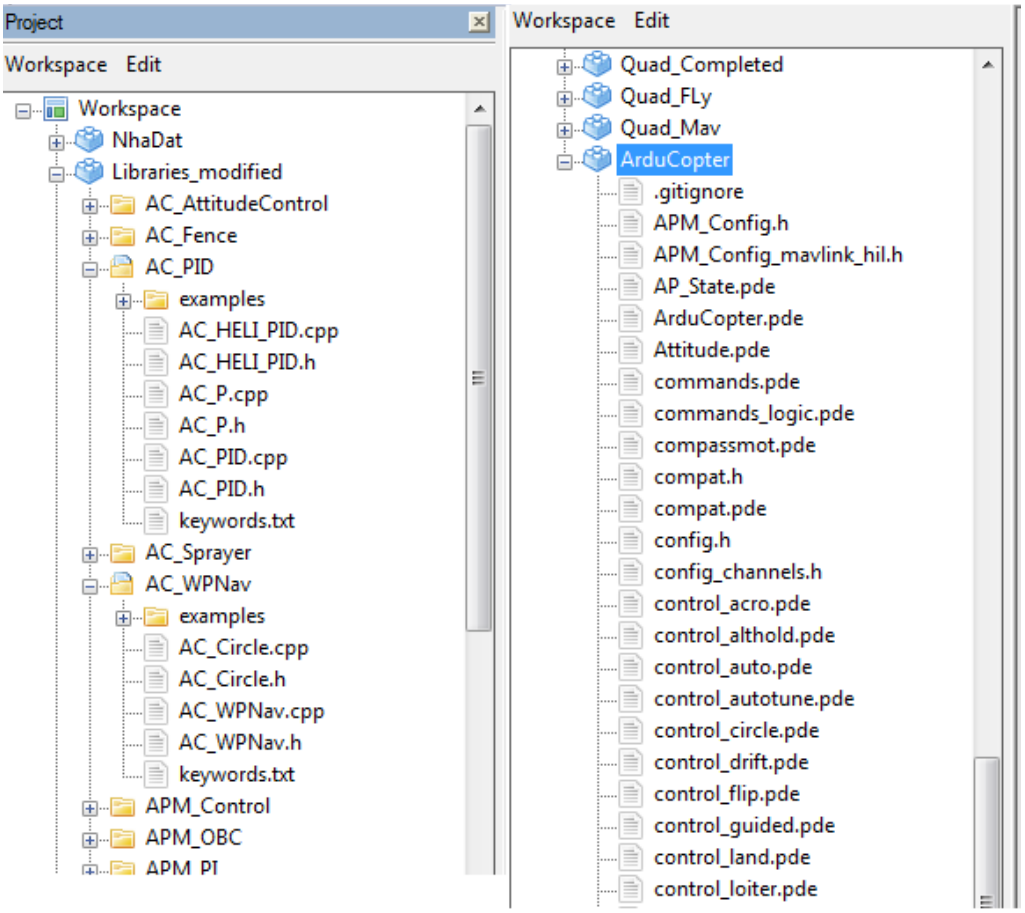


Figure B-1-4: ArduPilot Files

*.h files and *.cpp files are standard files for C/C++ programming language. In short, *.h files is the header files, includes any class name, function name, variables and constants. *.cpp is where the classes and functions in *.h are implemented. *.pde is the file format of Arduino language, which is the base language of ArduPilot, these files will be preprocessed into *.cpp files.

In the Libmaple – a source code with drivers for FlyMaple - there are some other files types such as ***.c** and ***.o**. The *.c is format of C language, meanwhile, *.o files are the result of compilation from a source code. They will be added into compile process by a **config.mk** file with the structure as followed.

Quadcopter_completed.pde | GCS_Mavlink.pde | GCS_FlyMaple_Common.cpp | esc_calibration.pde | config.mk

```
1  #config.mk START
2   # Select maple_RET6 for Flymaple
3   BOARD = maple_RET6
4   # HAL_BOARD determines default HAL target.
5   HAL_BOARD ?= HAL_BOARD_FLYMAPLE
6   # The communication port used to communicate with the Flymaple
7   PORT = /dev/ttyACM0
8   # You must provide the path to the libmaple library directory:
9   LIBMAPLE_PATH = $(HOME)/Learning_ArduPilot/libmaple
10  # Also, the ARM compiler tools MUST be in your current PATH like:
11  # export PATH=$PATH:~/Learning_ArduPilot/libmaple/arm/bin
12  #config.mk END
13
```

Figure B-1-5: config.mk structure pointing out the necessary drivers for FlyMaple [22]

In order to cross-compile, many rules, targets as well as building orders have to be defined. This structure is implemented into several *.mk files in mk directory of ArduPilot. From the definitions in these files, **figure B-1-6** and **figure B-1-7** can be used to present the "make" process in ArduPilot.

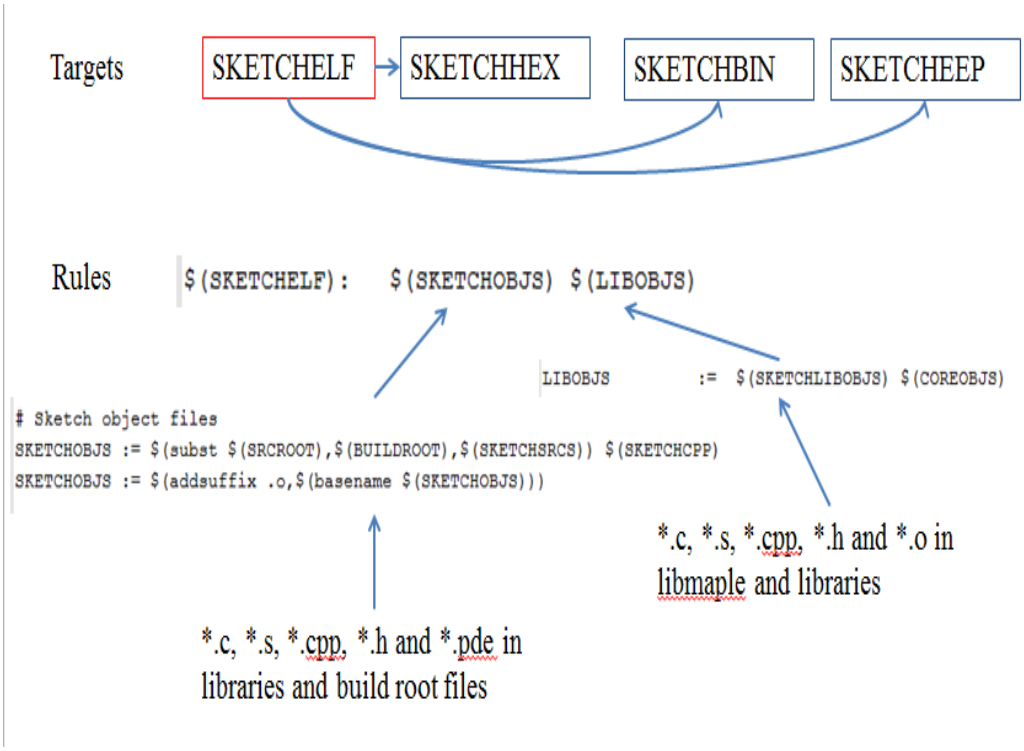


Figure B-1-6: Summary of "make" process by targets and rules

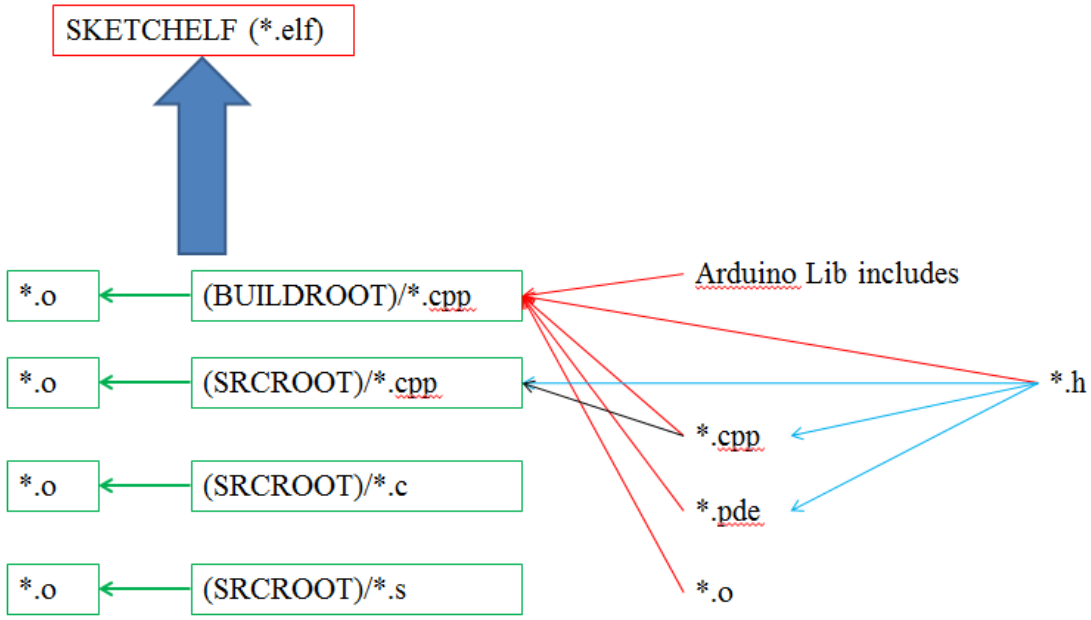


Figure B-1-7: Summary of "make" process by objects and file formats

These figures, along with **figure B-1-1**, give an overview look about GNU make process and the dependency of ArduPilot cross compiling, how the firmware is made. Knowing the basics of ArduPilot as presented is very important to handle

the problems of using ArduPilot with different hardware, in this case FlyMaple control board.

C.MAVLink protocol

The following information can be found in this appendix:

- Overview about Cyclic Redundancy Check
- MAVLink checksum calculation
- MAVLink supporting functions
- Generate C#, Java and Python using different tools

MAVLink Checksum

Cyclic Redundancy Check (CRC) is a very popular method used to detect error code implemented in many data transmission schemes [**48**]. By adding this checksum value at the end of every transmitting packet, for example the heartbeat packet mentioned above, receiver can check if the received message misses any frame during the transmitting process. Moreover, because of the variety of method to calculate this checksum, it can also be used as a tool to separate messages for different applications.

Cyclic Redundancy Check uses the **binary sequence** of the received message and the **polynomial number** to calculate the **remainder value** after several divisions. For example, if the receiver receives the message as [1110 0111 0110 1100], the polynomial is [1010] and we are waiting for a 3-bit CRC, the calculation for the remainder should be as described in **figure C-1-8**. As can be seen, the method used in this calculation is XOR (exclusive OR), where 1 + 1 = 0; 0 + 0 = 0; 1 + 0 = 0 + 1 =1.

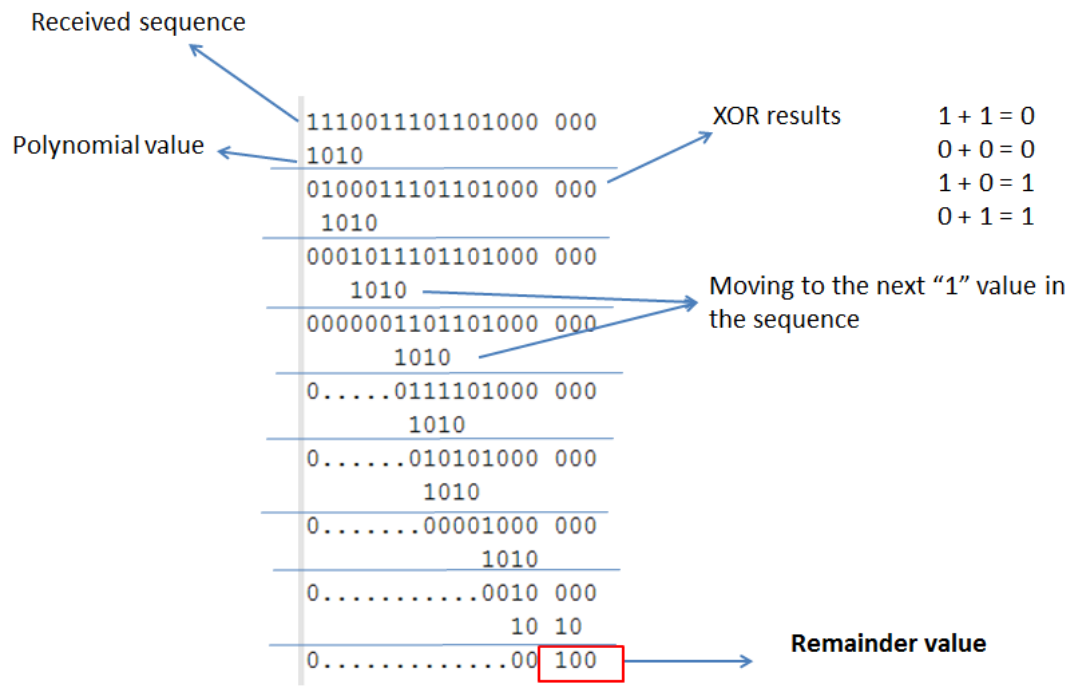


Figure C-1-8: A simple Checksum calculation

Now if we replace the reserved place at the right with the true **Remainder Value** (checksum value), the result would be zero:

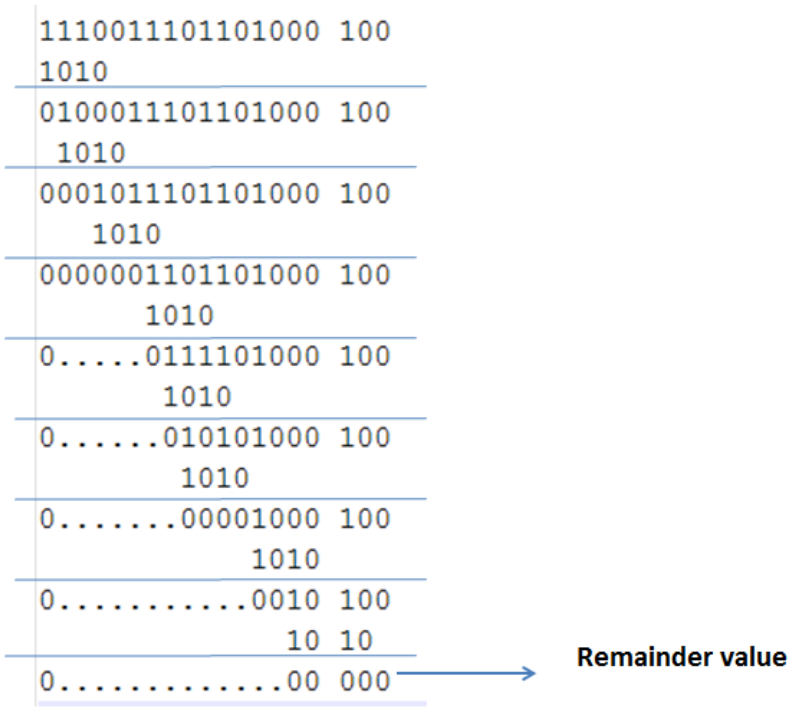


Figure C-1-9: Remainder value is Zero as message included CRC

When the message including the CRC is transferred and then loses a bit during transmitting due to noise or other reasons, the receiver will notice the difference

by encoding the message and find the remainder value. Nowadays, with the development of telecommunication and transmitting protocol, there are various standards for checksum such as CRC-4, CRC-8, CRC-16, CRC-32, etc. Each of them has their own usages and applications. More details about these checksum calculations can be seen in [**25**] and [**48**].

MAVLink Checksum

MAVLink uses CRC-16-CCITT standard, which is used in various communication protocol such as X.25, Bluetooth, etc [**25**]. The result of this checksum calculation is a 16 bits remainder value, this value is then put into the packet as CRCs as shown above. For example, the checksum value for the heartbeat message in figure 1-3 after calculating is [0001 1000 0001 0011] in binary, which is 1813 in hexadecimal. This value is then transfer as 18 in CKA and 13 in CKB.

However, from version 1.0, MAVLink developers have made a small change in the calculation of the checksum value. In order to avoid the phenomenon that *"two devices using different message versions incorrectly decode a message with the same length"* [**49**], which occasionally caused problems in version 0.9, an extra CRC calculation step is added into this protocol. As they mentioned in [**49**], now every message will have an extra constant beside the message ID called "the seed". After calculating checksum for every byte in the packet, MAVLink will have one more calculation with this "seed", and then the value of CKA and CKB will be this value. These seeds are defined in the library of MAVLink, for instances, **figure C-1-10** introduces the seeds in Ardupilotmega.h. Because Heartbeat message ID is 0, its seed would be 50 as can be seen

```
#ifndef MAVLINK_MESSAGE_LENGTHS
#define MAVLINK_MESSAGE_LENGTHS {9, 31, 12, 0, 14, 28, 3, 32, 0, 0, 0, 6, 0, 0,
#endif

#ifndef MAVLINK_MESSAGE_CRCS
#define MAVLINK_MESSAGE_CRCS {50, 124, 137, 0, 237, 217, 104, 119, 0, 0, 0, 89,
#endif
```

Figure C-1-10: Seeds for extra CRC calculations in Ardupilotmega.h

Supporting functions in MAVLink libraries

Beside the definitions for messages, MAVLink library also provide users with some useful functions to implement it into different projects. A simple example for these functions is the checksum calculation, it can be found in **checksum.h** as shown below.

```
static inline void crc_accumulate(uint8_t data, uint16_t *crcAccum)
{
        /*Accumulate one byte of data into the CRC*/
        uint8_t tmp;

        tmp = data ^ (uint8_t)(*crcAccum &0xff);
        tmp ^= (tmp<<4);
        *crcAccum = (*crcAccum>>8) ^ (tmp<<8) ^ (tmp <<3) ^ (tmp>>4);
}
```

Figure C-1-11: Function to calculate checksum in MAVLink library

These supporting functions are provided in C/C++ language; however, there are tools to convert this library into Python, C# and Java. **Figure C-1-12** and **table C-1-3** gives a summary about these tools. (Should put these info in Apendix B)

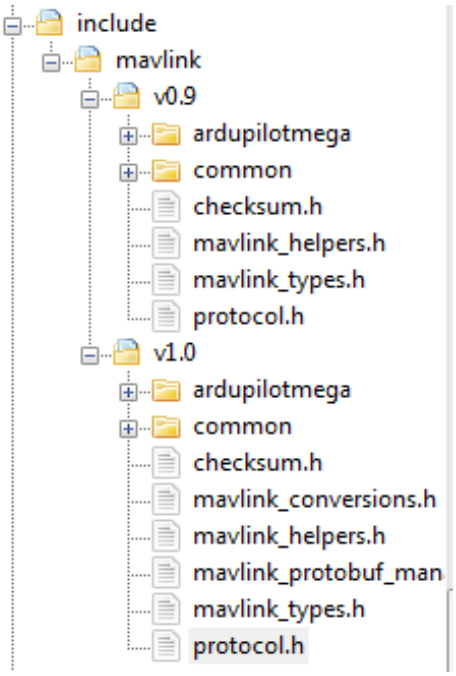


Figure C-1-12: Files contain supporting functions in MAVLink v0.9 and v1.0

Table C-1-3: Supporting functions in MAVLink library

Function group	Reference files	Functions	Application
Messages handling group	mavlink_helpers.h	mavlink_parse_char	Convenient function to handle the received messages
Messages sending group	mavlink_helpers.h protocol.h	mavlink_finalize_message_chan	Convenient function to send message and calculate checksum
		mavlink_finalize_message	Convenient function to send message and calculate checksum
		_mavlink_send_uart	Send message
		_mav_finalize_message_chan_send	Convenient function to send message and calculate checksum
		_mavlink_resend_uart	resend a message in the same

			channel
		mavlink_msg_to_send_buffer	Pack a message and send it via a byte stream
		mavlink_msg_get_send_buffer_length	Get the required buffer size for this message
Channels handling group	protocol.h mavlink_helpers.h	mavlink_get_channel_status	Check the status of the channel (number of available free space)
		mavlink_get_channel_buffer	Check the buffer status in the channel (number of available free space)
		mavlink_reset_channel_status	Reset the status of the channel
Checksum group	mavlink_helpers.h checksum.h	mavlink_start_checksum	Return the init value 0xffff
		crc_init	Return the init value 0xffff
		crc_calculate	Return the checksum over a buffer bytes
		crc_accumulate	Return the checksum over a transferred byte
		mavlink_update_checksum	Return the checksum over a transferred byte
		crc_accumulate_buffer	Return the checksum over an array of bytes
Buffer handling group	protocol.h	mav_array_memcpy	Copy the values from the pointed location to the destination (buffer)
		_mav_put_char_array	Place a char array into a buffer
		_mav_put_int8_t_array	Place a int8_t array into a buffer
		_MAV_RETURN_char_arr	Get a char from

		ay	buffer
		_MAV_RETURN_uint8_t_array	Get a uint8_t from buffer
		_MAV_RETURN_int8_t_array	Get a int8_t from buffer
Values conversion group	mavlink_conversions.h	mavlink_quaternion_to_dcm	Convert a quaternion to a rotation matrix
		mavlink_dcm_to_euler	Converts a rotation matrix to euler angles
		mavlink_quaternion_to_euler	Converts a quaternion to euler angles
		mavlink_euler_to_quaternion	Converts euler angles to a quaternion
		mavlink_dcm_to_quaternion	Converts a rotation matrix to a quaternion
		mavlink_dcm_to_quaternion	Converts a rotation matrix to a quaternion

Working procedure of supporting functions

As can be seen, MAVLink library provides developers with many convenient functions to speed up process of establishing a communication system with MAVLink protocol. Among them, **mavlink_parse_char** and can be considered as one of the most important function. Knowing how it work will understand the listening structure of MAVLink and simplify the process of integrate MAVLink protocol into a project.

Table C-1-4: mavlink_parse_char function

Function	**mavlink_parse_char** (uint8_t chan, uint8_t c, mavlink_message_t* r_message, mavlink_status_t* r_mavlink_status)		
Inputs	**uint8_t**	chan	Channel receives the

			message
	uint8_t	c	character (byte) to parse
	mavlink_message_t*	r_message	packet anatomy
	mavlink_status_t*	r_mavlink_status	status flag to control the function
Outputs	**uint8_t**	return 0 if no message could be decoded and 1 in the other case	

When a byte value is read by the serial port, **mavlink_parse_char** will be called to decide what the system should do next. The general idea about this function can be explained by **figure C-1-13.**

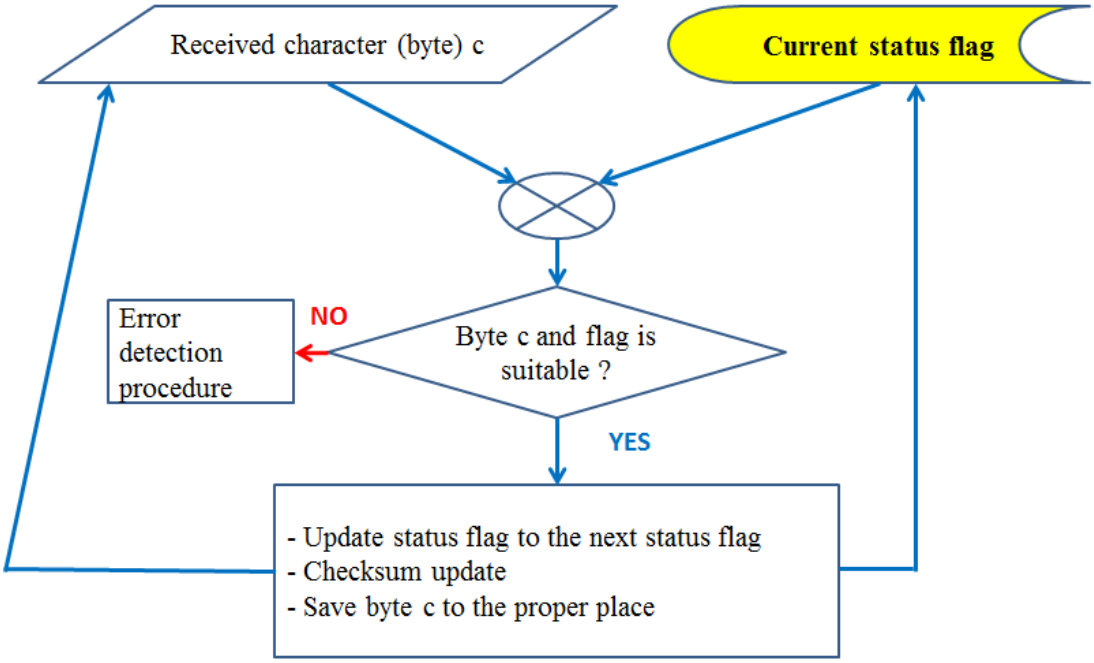


Figure C-1-13: mavlink_parse_char general procedure

After receiving a byte, depending on the status flag, the control system will decide what to do with that byte. If the byte and the flag are compatible, the handling process for that flag will be triggered. Although this procedure has some small differences based on the flag, it always includes three main functions: **update the flag**, which will be suitable for the next coming byte; **update checksum**; **save received byte** to memory. After this procedure,

mavlink_parse_char is finished and then go back waiting for the next byte. For instance, **figure C-1-14** describes the procedure when the STX value comes.

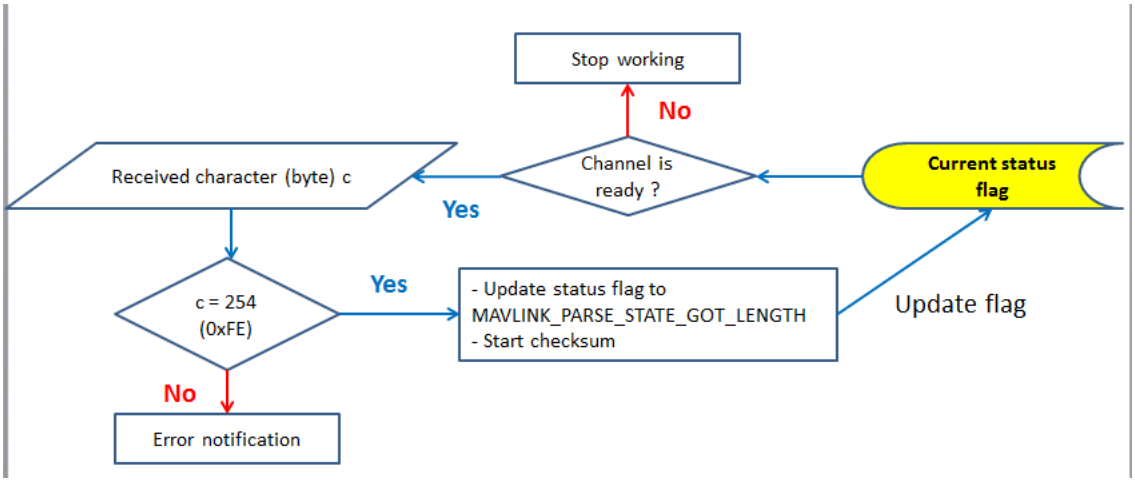


Figure C-1-14: : A detail procedure for mavlink_parse_char

In **figure C-1-14**, MAVLINK_PARSE_STATE_GOT_LENGTH is the next status flag, which will trigger the process handling the next incoming LEN frame. As can be seen, programmer only need to generate function to read the message and then call this function to handle the message, byte by byte, every received frame will be put in right place and waiting for the handling process.

Tools to create MAVLink libraries in C#, Java and Python

By default, MAVLink protocol is a group of header files in C code. However, in an effort to make this protocol become familiar with many other developing project, the developers of MAVLink has created a simple tools to convert from C other programming language. Figure B-8 introduce MAVLink in a **d**ynamic **l**ink **l**ibrary (dll) file, which can be used in a project using C# as the programming language.

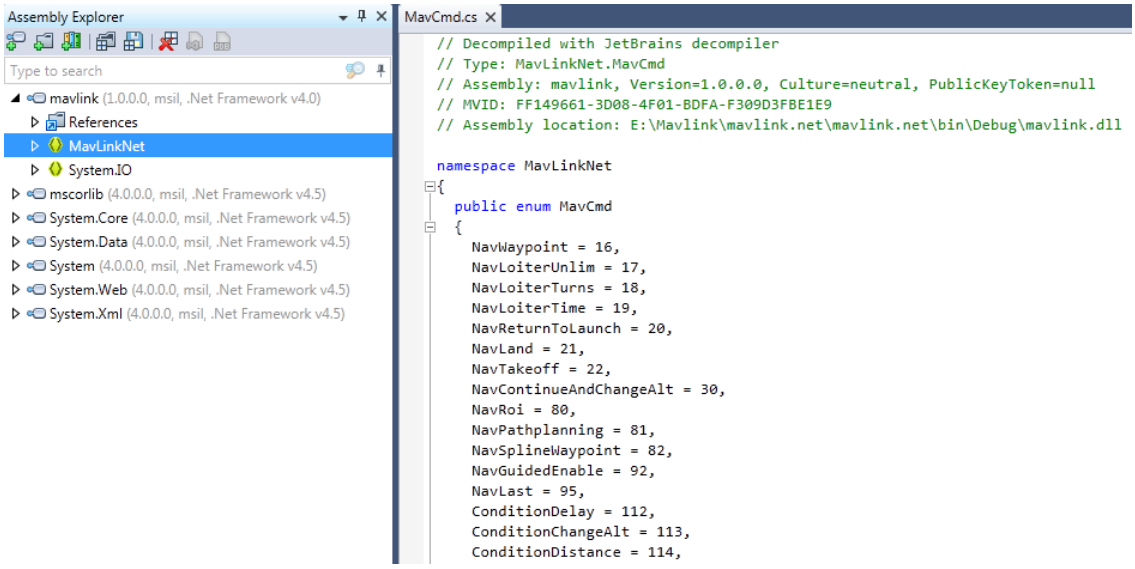


Figure C-1-15: MAVLink in a dll file

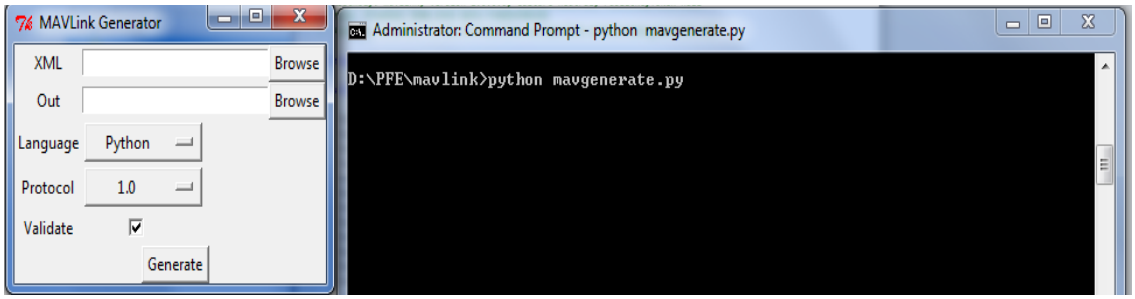


Figure C-1-16: MAVLink Generator and command to call it in prompt

Besides the original tool, there are some other open-source project which can be used to create necessary library files of MAVLink for other programming language such as **MAVLink.net**, more information about this project as well as the MAVLink Generator can be found in [**50**] and [**51**].

D.Ground Control Stations comparison

As mentioned, MAVLink protocol has become popular and has been used worldwide in several GCS. In this appendix, three different GCS using MAVLink protocol will be presented, as well as their pros and cons. From these discussions, this report will point out the most suitable GCS for using in this project.

QGroundControl

QGroundControl is an Open Source Mircro Air Vehicle Ground Control Station/ Operator Control Unit, which is based on PIXHAWK's Groundstation project [**52**].

There are several advantages of this GCS. QGroundControl is multi-platform, which means it is suitable for Windows, Linux and MacOS operation systems. This GCS is also an open-source project which developers can change build a personal version of it. More importantly, this GCS can support more than two hundred vehicles in parallel; therefore, it is suitable for projects requiring multiple unmanned systems working together.

Additionally, the interface of QGroundControl is the most friendly among the three, it even includes the voice report to attract the attention of users in specific cases and a communication console in which users can see the hex signal sent by the autopilot system. This signal display is very important in case of debugging for a testing firmware.

Figure D-1-17: QGroundControl interface

However, there is one major problem with this GCS. After several tests, using both sample code [**24**] and ArduPilot code to test the communication with

FlyMaple controller board, it is clear that FlyMaple cannot receive the messages sent by this GCS. It is not the problem that FlyMaple does not understand these messages. In fact, FlyMaple does not receive any character or byte from QGroundControl. Since FlyMaple still can receive and understand messages from the other two GCS, this problem can only be explained either by the fact that QGroundControl is not compatible with FlyMaple, even though it still can receive message from this board and understand the MAVLink protocol in that message perfectly; or by the fact that there is some undocumented configuration options that I missed.

Although there are many advantages, the problem in communication discards QGroundControl as the Ground Control Station for the quadricopter. Nevertheless, QGroundControl is convenient for debugging and testing the sending function of the controller board, therefore, many tests in this project have used this GCS as a testing environment for descending traffic, as can be seen from the other parts in this report.

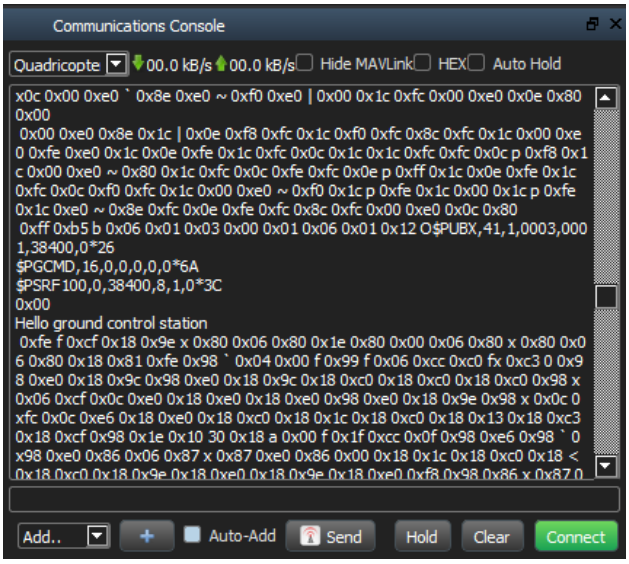


Figure D-1-18: Debugging with signal received display in QGroundControl

Mission Planner

According to its homepage [**53**], *"Mission Planner is a free, open-source, community-supported application developed by Michael Oborne for the open-*

source APM autopilot project", it is also developed and supported by APM, the same developing group as ArduPilot. As a result, this GCS is fully compatible with autopilot systems using ArduPilot firmware. The interface of this program is also user friendly, as can be seen in **figure D-1-19**, it also contains many pros as QGroundControl (multiple-system-paralleling control, the ability to use in simulation, etc).

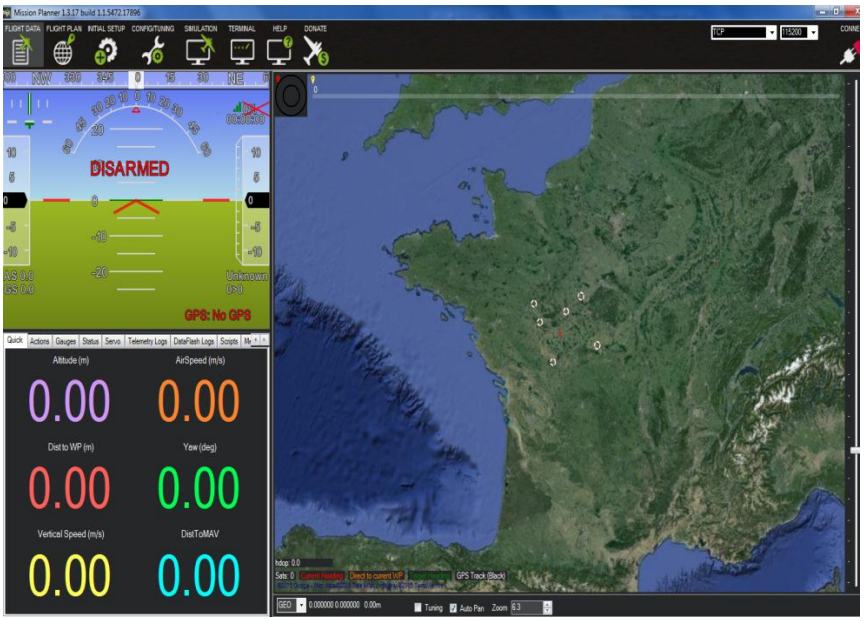


Figure D-1-19: Mission Planner interface

Nevertheless, Mission Planner has its own drawbacks. The first inconvenience of this GCS is that it does not show the hex signal it received from the autopilot. Although Mission Planner provides users some space to display the text messages it received, this place only displays the text, not the whole package hex value like QGroundControl. The second problem with Mission Planner is that it is Windows compatible only, which is a major minus comparing to other GCS supporting multiple Operation Systems. However, this GCS has passed multiple tests for FlyMaple, which examines not only the possibility to receive the messages from the control system via Wi-Fi connection but also the ability to send the messages to the board and receive the responses from the board perfectly. For the objective of communication via TCP protocol with FlyMaple, Mission Planner is obviously the best solution so far.

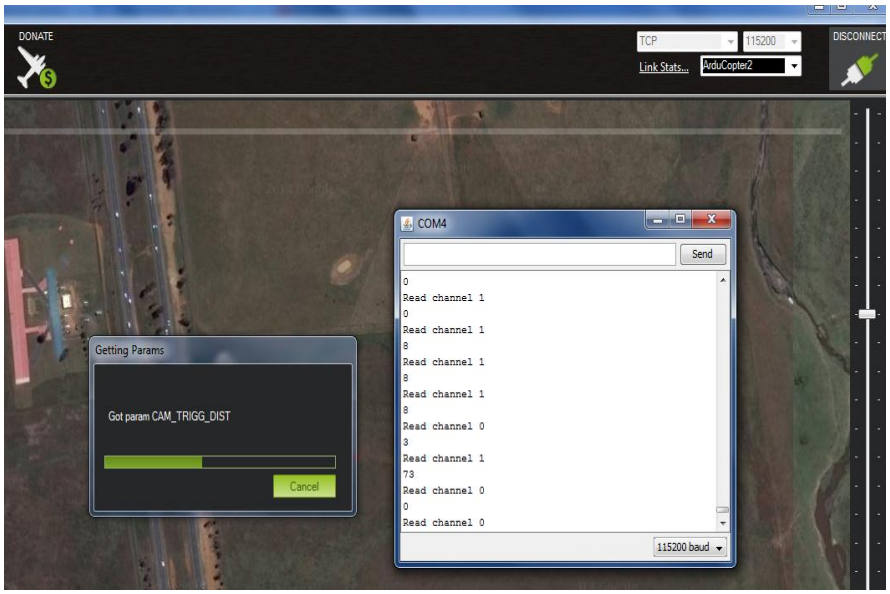


Figure D-1-20: Parameters transferring test with Mission Planner

APM Planner 2.0

APM Planner 2.0 is the combination between the Mission Planner and QGroundControl. It is believed to have the pros of both older GCS program: *"APM Planner 2.0 is the next generation of ground control station. It is the offspring of Mission Planner and QGround Control, combining the simple user interface of Mission Planner and cross platform capability of QGround Control."* [**54**]

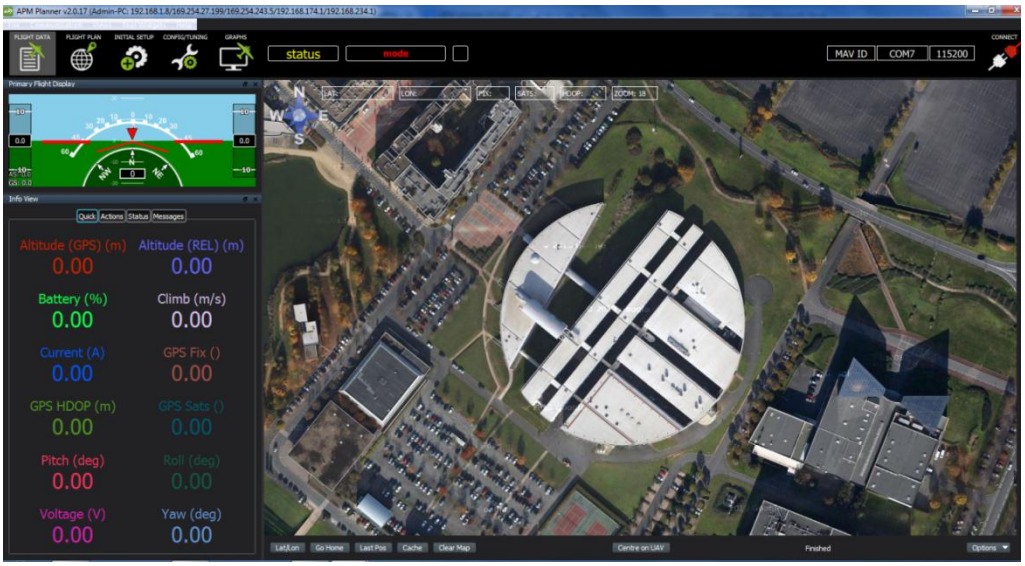


Figure D-1-21: APM Planner 2.0 interface

Although it can send and receive messages in certain cases, this GCS failed in the test of changing control parameters, one of the most basic task while using autopilot. In this case, it is also not as good as the Mission Planner above. Nevertheless, with the ability to communicate in some situation, this GCS will be used as a double-check for Mission Planner, in particular parameters changing tests and mission reading/writing tests.

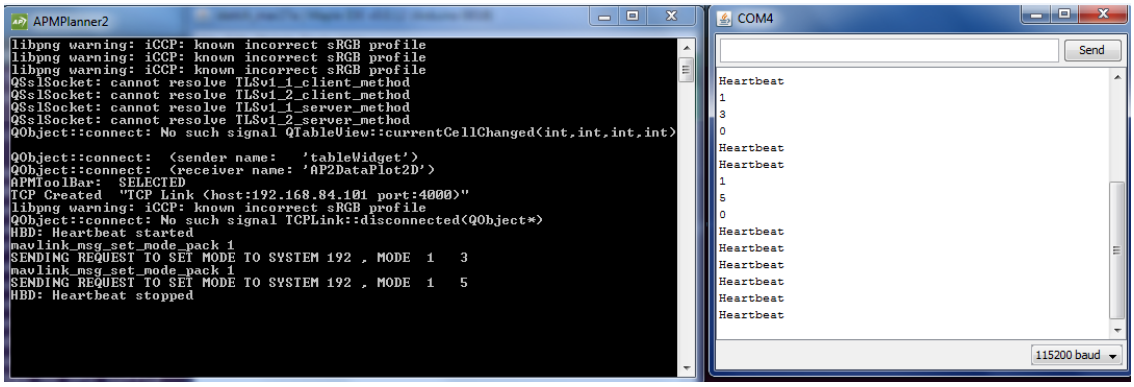


Figure D-1-22: Changing mode test with APM Planner 2.0

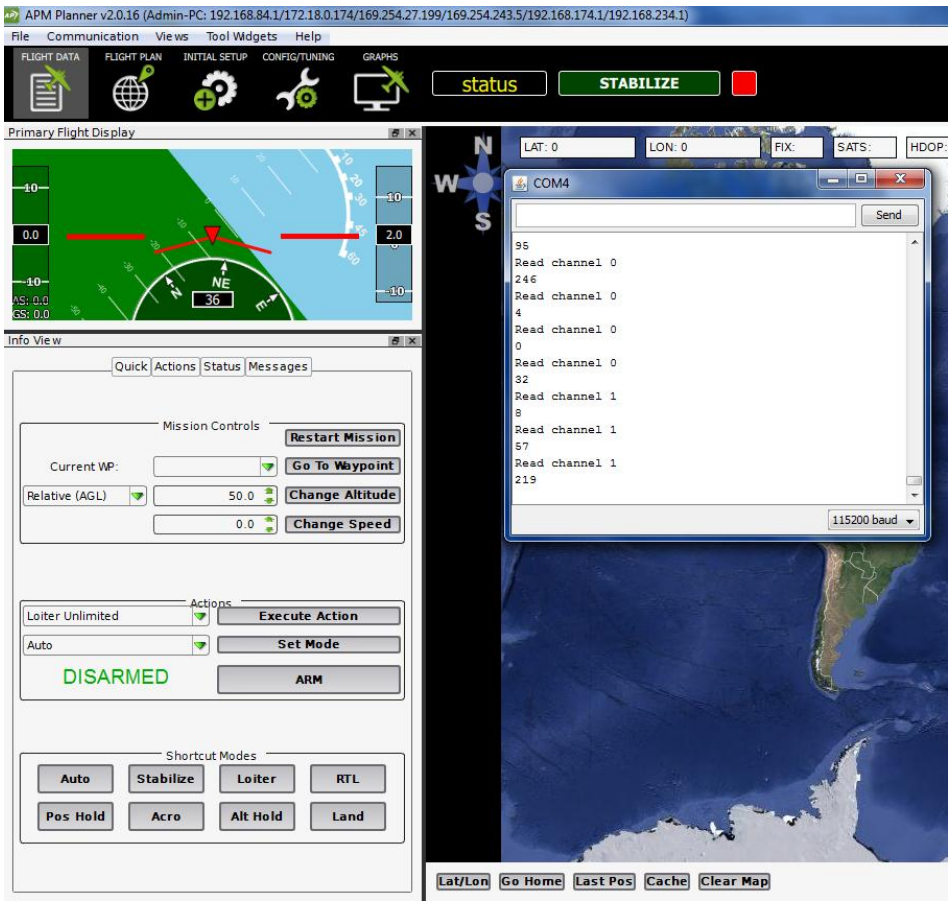


Figure D-1-23: A test with attitude and sensors using APM Planner 2.0

A summary of the pros and cons can be seen from the table below. To sum up, **Mission Planner** is the best solution for this project and will be the main GCS in real fly testing. **QGroundControl** is suitable for basic tests and debugging, meanwhile, **APM Planner** will be in charge of function tests to see if the results of Mission Planner are correct.

Table D-1-5: Comparing 3 Ground Control Stations

Test results and properties of GCS	**QGroundControl**	**Mission Planner**	**APM Planner 2.0**
Easy to learn	√	√	
Documentation available	√	√	
Open source	√	√	√
Windows compatible	√	√	√
Linus/MacOS compatible	√		√
Command display	√		√
Receiving Messages	√	√	√
Sending Messages		√	√
Display hex code received from autopilot systems	√		
Parameters loading		√	√
Advance commands		√	
Suggesting function	**Debugging**	**GCS for real flight**	**Double check for Mission Planner**

Tests with GCSs

In communication between autopilot system and Ground Control Station, changing control parameters and reading/writing missions is very important since without them, autopilot system cannot fulfill its objectives.

- **Changing control parameters**

In ArduPilot, there are more than 300 control parameters. These parameters are related to PID calculating, to sensors sensitivity, to radio controlling, etc. As indicated above, among three GCSs using MAVLink protocol, Mission Planner

is the only one that can handle totally these commands. A simple test was done using Mission Planner and double checked by APM Planner 2.0

In order to change control parameters in Mission Planner, users can use CONFIG/TUNNING tab, and then choose the constant that need to be changed.

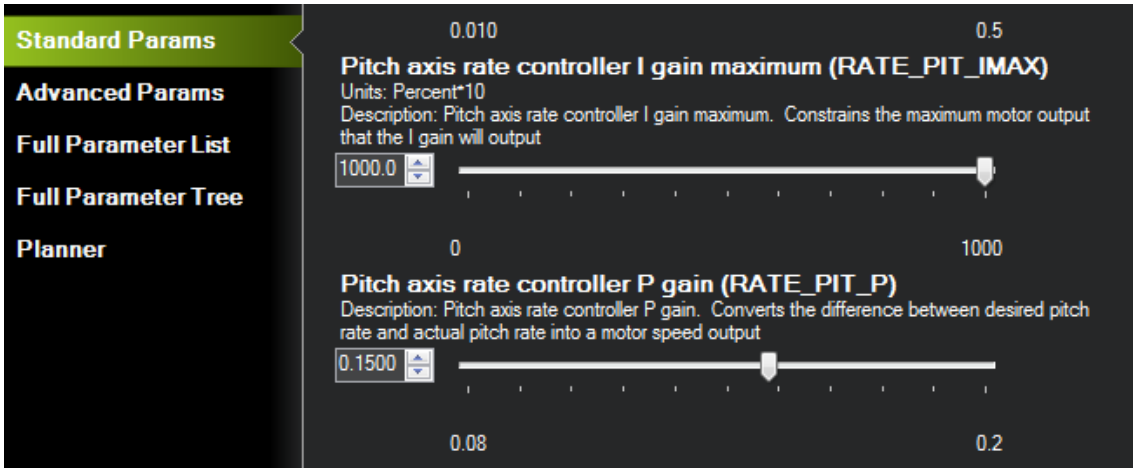


Figure D-1-24: Parameters displays in Mission Planner

After changing the value, user needs to click **write** it into autopilot system. A message to define this new value will then be sent to autopilot and if the procedure is successful, Mission Planner will display a notification as shown in **Figure D-1-25**.

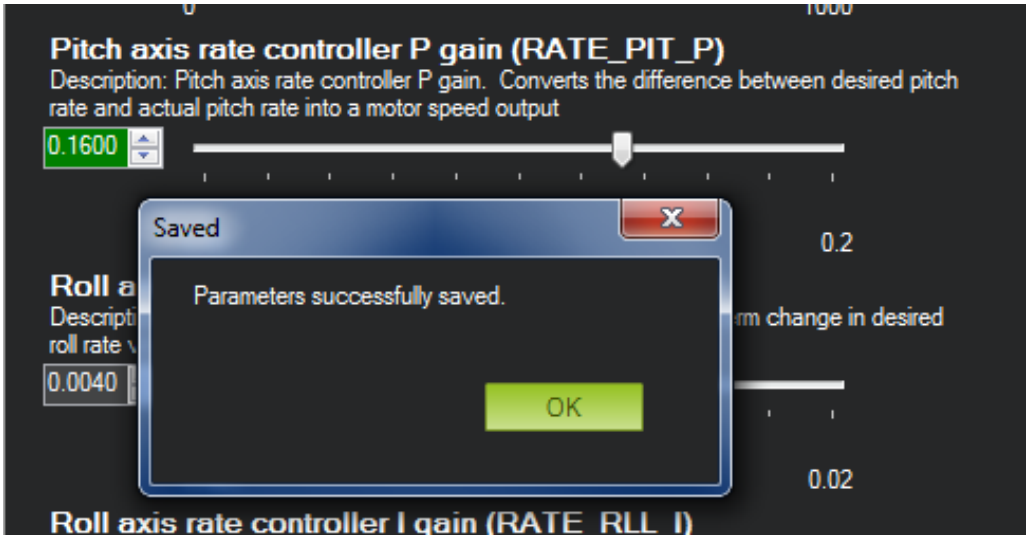


Figure D-1-25: Notify in Mission Planner after Ack message received

This value is then checked again by loading it using both Mission Planner and APM Planner 2.0. In both case, the value of **RATE_PIT_P** is 0.16, which is the confirmation for the test.

Figure D-1-26: Checking result with APM Planner 2.0

- **Flight plan design and writing/reading flight plan from GCS.**

In general, there are various ways to create a flight plan for an UAV. Users can use supporting programs to create this plan or use some GCS supporting this application and can communicate with autopilot system such as Mission Planner or APM Planner 2.0. Moreover, flight plan is nothing but a *.txt file containing information about GPS longitude, latitude, altitude of the waypoint…Therefore, users can create this file with a simple text editor. **Figure D-1-27** presents a flight plan created by Mission Planner.

```
1  QGC WPL 110
2  0   1   0   16  0   0   0   0   46.660964   0.362056    50.000000   1
3  1   0   3   16  0.000000    0.000000    0.000000    0.000000    46.661079   0.362527    100.000000  1
4  2   0   3   16  0.000000    0.000000    0.000000    0.000000    46.660886   0.362417    100.000000  1
5
```

Figure D-1-27: A flight plan with three waypoints

After creating the plan, it must be uploaded into the controller board. In ArduPilot, as long as the board is fully supported by the developed firmware,

writing a mission into autopilot system is nothing more than a communication sequence as shown earlier. **Figure D-1-28** indicates this process.

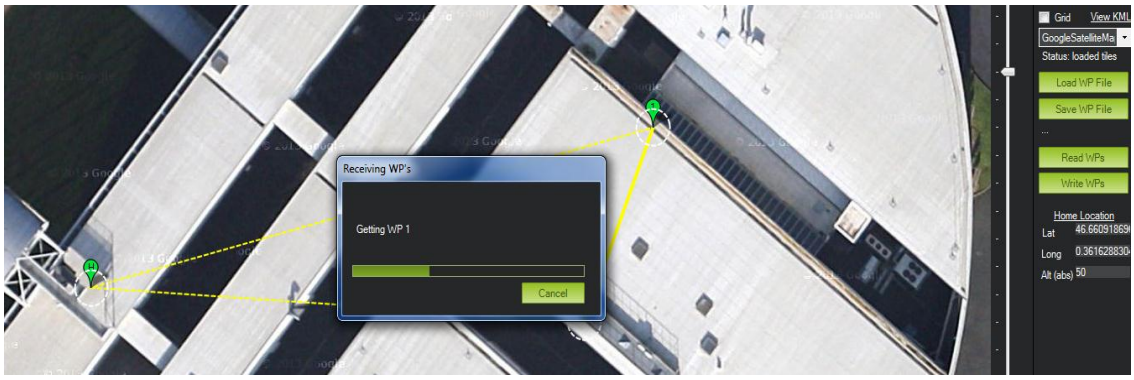


Figure D-1-28: Reading flight plan

Figure D-1-29: Writing flight plan

The two tests above are just some of the tests have been done to check the compatibility between ArduPilot, FlyMaple and Ground Control Station. The results from these tests led to the conclusion about the GCS in **table D-1-5** above.

E. Electronic Speed Controller calibration and programming

Calibration

In general, the calibration process can be done with the following steps:

- Send the maximum PWM values.
- Wait for the confirming tone from the ESC, after a few seconds, send the minimum PWM values.

- A different tone will confirm the calibration process has been done successfully.

Procedure to calibrate the ESC using the specific firmware built by ArduPilot framework:

- Compile and upload the firmware into the board.
- Connect the ESC needed to calibrate and the control board, check if the leg of the output pin and the wire of the ESC has been connected correctly. **DO NOT CONNECT THE RED WIRE OF THE ESC WITH THE BOARD**.
- Power the board with the USB port from the computer.
- Wait a few second until see the confirmation in the console window in the computer. The ESC is now receiving the maximum PWM value.
- Power the ESC with the battery and wait for the tone.
- Enter a character into the console window. The minimum PWM value will be sent. There will be a tone inform that the procedure has been done correctly.

Programming

In fact, the programming procedure of the ESC is nothing different than the calibration. In case of not sending the minimum value after a some second sending the maximum one, the ESC will automatically enter the programming mode. Using the tone signals, users can define which setting is in programming. To accept the new settings, send the low PWM value, wait for the conforming tone and then unplug the battery. The ESC can only be programmed one setting at a time, so this procedure can be done slowly but effectively.

DO NOT CONNECT THE RED WIRE OF THE ESC WITH THE BOARD

F. Simulation settings and other results

Simulation settings

The following values has been set up for the simulations

Table F-1-6: Some settings for the simulation

Parameters	Values	Meaning
FRAME	0	Type of frame of the multicopter, "0" represent quadricopter + frame
MAG_ENALBE	1	Enable using of magnetometer data
FS_THR_ENABLE	1	Enable throttle failsafe
BATT_MONITOR	4	Enable values from the battery sensors, getting total battery voltage and current
COMPASS_LEARN	0	Disable the automatic learning of compass offsets
COMPASS_OFS_X	5	Compass offset value in x-axis
COMPASS_OFS_Y	13	Compass offset value in y-axis
COMPASS_OFS_Z	-18	Compass offset value in z-axis
SIM_GPS_DELAY	2	Delay time (in second) of the GPS signal
SIM_ACC_RND	0	Acceleration noise (in m/s/s)
SIM_GYR_RND	0	Noise for the data from the gyroscope
SIM_WIND_SPD	20	Speed of wind (in m/s)
SIM_WIND_TURB	5	The turbulence of the wind
SIM_BARO_RND	0	Noise for the data from the barometer
SIM_MAG_RND	0	Noise for the data from the magnetometer
INS_ACCOFFS_X	0.001	Offset for accelerometer in x-axis
INS_ACCOFFS_Y	0.001	Offset for accelerometer in y-axis
INS_ACCOFFS_Z	0.001	Offset for accelerometer in z-axis
INS_ACCSCAL_X	1.001	Scale value for accelerometer in x-axis
INS_ACCSCAL_Y	1.001	Scale value for accelerometer in y-axis
INS_ACCSCAL_Z	1.001	Scale value for accelerometer in z-axis

Other parameters can be found in the copter-params.parm file in the autotest directory. For more information about the parameters in simulation and how to use them, please refer this site [**55**].

Simple Integral Backstepping control algorithm gains

Control block	Gain	Value
Altitude control	Constant	22.36
	Tca1	0.05
	Tca2	0.02
	Tca3	0.01
	ld4	0.01
	m	1.14
	c7	3.5
	c8	1.5
Attitude control	a1	(-1.125)
	a2	0.006971153846154
	a3	1.116071428571428
	a4	0.006473214285714
	a5	(-0.034934497816594)
	b1	96.153846153846146
	b2	89.285714285714292
	b3	43.668122270742359
	Tcr1	0.01
	Tcr2	0.05
	Tcr3	0.01
	Tcp1	0.01
	Tcp2	0.05
	Tcp3	0.01
	Tcyw1	0.01
	Tcyw2	0.05
	Tcyw3	0.01
	c1	15
	c2	6
	c3	15
	c4	6
	c5	10
	c6	5
	ld1	0.07
	ld2	0.05
	ld3	0.01

Other simulation results

This part introduces the comparisons between controllers in simulation, which have not been shown in the report above.

1. Flight plan 2 simulation tests

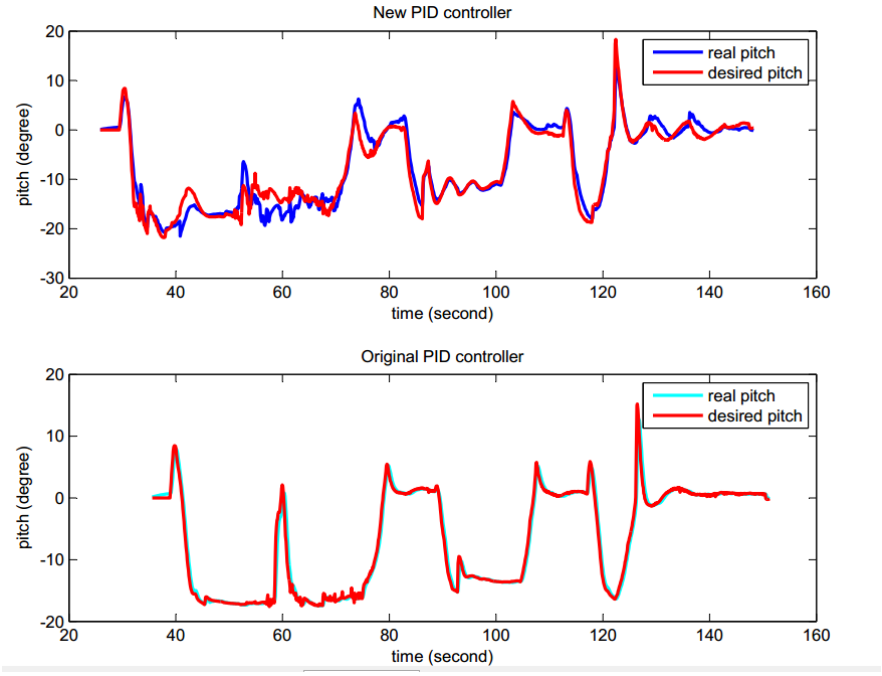


Figure F-1-30: Real pitch and desired pitch with the new and the original PID controller

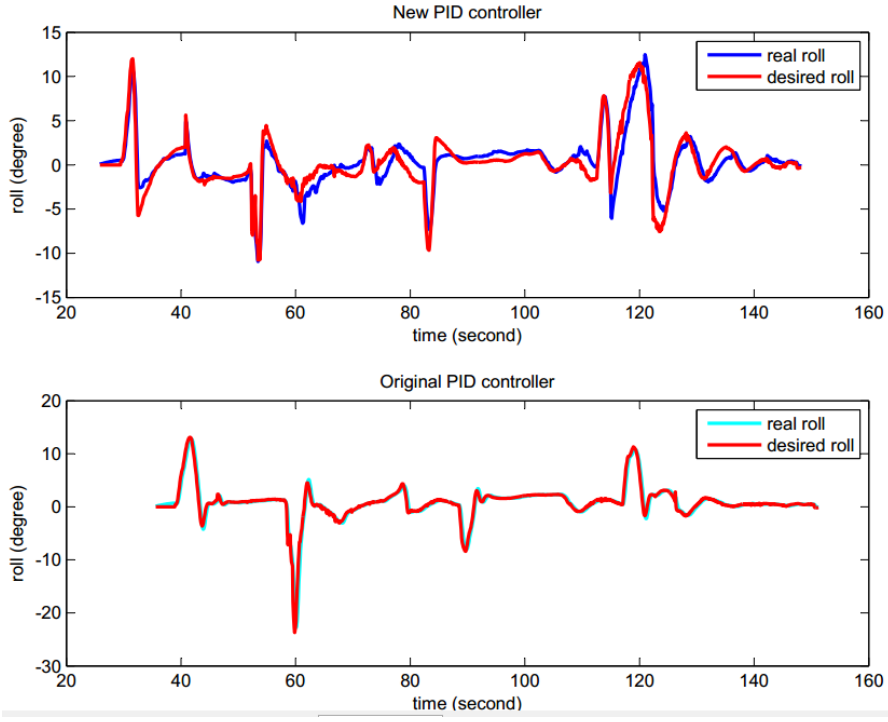


Figure F-1-31: Real roll and desired roll with the new and the original PID controller

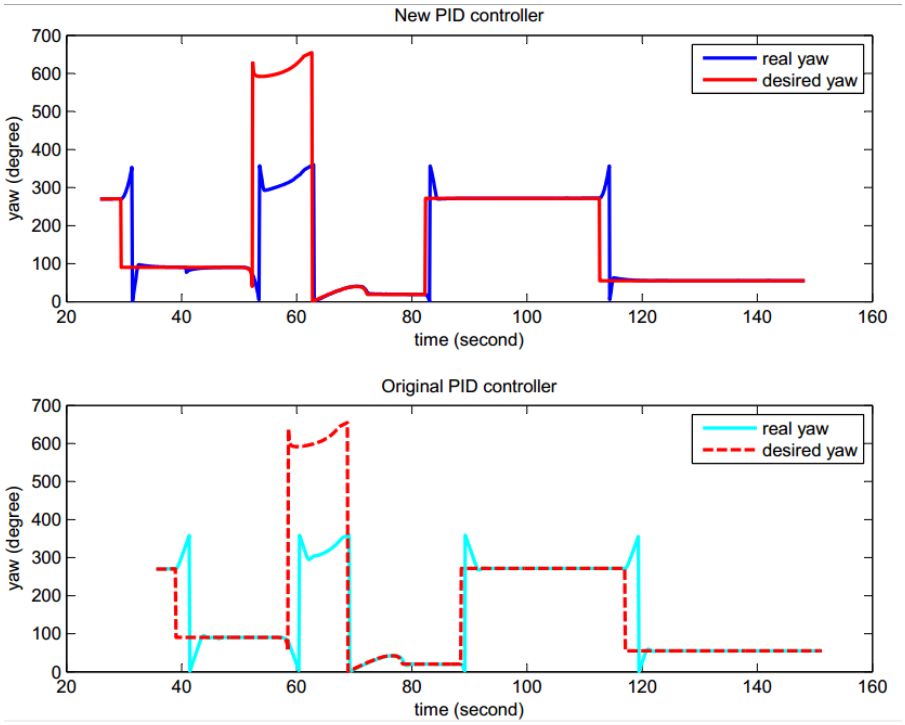


Figure F-1-32: Real roll and desired roll with the new and the original PID controller

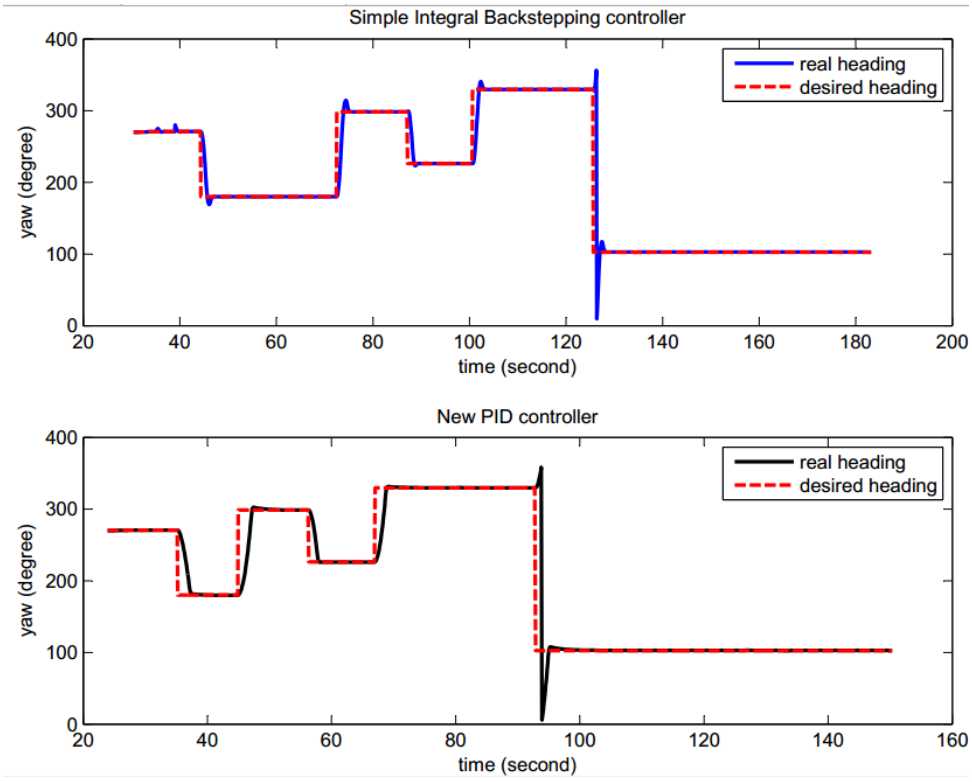


Figure F-1-33: Yaw and desired yaw of simulation with mission 1 using simple IB controller and new PID controller