Politecnico di Torino

Master of Science Degree in COMPUTER ENGINEERING

Master Thesis

Implementation of an autopilot for UAV/UGV systems based on Android smartphone

Supervisor:
prof. Marcello Chiaberge

Candidate:
Eros Giorgi
S238145

10 April 2019

Abstract

In the last year, interest from people about drones and in particularly about UAVs (Unmanned Air Vehicles), vehicles without an onboard human presence, has had great growth. What at the beginning of the 1900s was only a field of military interest, in recent year it has received the attentions of various sectors of society, among which civil area, commercial area, scientific area, agricultural area etc, with many scope of applications.
So, the main idea followed has been thos of the development of ligth aircrafts whit reduced dimension and with minimum architectural complexity.
One of the principal components for this aircrafts is the software that controls the drone's activity.
It is called autopilot.

The purpose of this thesis project is to design and implement an Android application for high-end smartphone that acts as an autopilot and, through some control laws, it can move the drone.
In particular will be discussed the developement of control and command software of an UAV quadrotor, trying to use in the best way the smartphone's architecture, including the sensoristic part such as accelerometer, gyroscope, barometer, magnetometer, and other systems like GPS system, WiFi system, camera system etc.
Work with these sensors the software will in charge of controls drone's attitude and altitude in the first, and then will be able to perform maneuvers given remotely by a pilot using radio.

Another purpose of this project is to manage the connection and comunication with the GCS (Ground Control Station), a base station to the ground, through the smartphone's WiFi module, in order to transmit some data like telemetry data, GPS informations, video streaming and other informations about the UAV.

Sommario

Negli ultimi anni, l'interesse verso il mondo dei droni ed in particolare verso gli UAVs (Unmanned Air Vehicles), veivoli senza pilota a bordo, ha avuto un crescita sostanziale da parte della società. Ciò che all'inizio del '900 era solo un campo di interesse militare, negli ultimi anni ha avuto l'attenzione da parte di vari settori della società, tra le quali il settore civile, commerciale, scientifico, agricolo etc, con una vasta possibilità di applicazioni.
Lo sviluppo quindi di veivoli sempre più leggeri, dalle dimensioni ridotte e con una minor complessità architetturale è stata una delle principali idee perseguite.
Una delle componenti principali per questi tipi di veivoli è il software che regola il controllo ed il movimento del drone.
Esso prende il nome di autopilota.

Questo progetto di tesi si prefige lo scopo di progettare e implementare un'applicazione Android per smartphone di alto livello che agisca da autopilota e, tramite alcune leggi di controllo, possa governare un drone.
In particolare verrà discusso lo sviluppo del software di controllo e di comando di un UAV quadrimotore, cercando di sfruttare al meglio l'architettura di uno smartphone, compresa la parte sensoristica quali accelerometro, giroscopio, barometro, magnetometro, e altri sistemi quali sistema GPS, sistema WiFi, fotocamera etc.
Sfruttando i suddetti sensori, il software sarà quindi in grado di controllare la posa e la quota del drone in prima battuta, ed in seguito sarà in grado di eseguire delle manovre impartite a distanza da un pilota tramite il Radio Comando (RC).

Il secondo obiettivo prefissato dal progetto è quello del collegamento e della comunicazione con la GCS (Ground Control Station), la stazione a terra, sfruttando il modulo WiFi integrato sullo smartphone, al fine di trasmettere i dati telemetrici, le informazioni GPS, lo streaming video e altre informazioni di controllo del UAV.

Acknowledgements

Anche se potrebbero non bastare poche parole per ringraziare tutte le persone che mi hanno donato affetto e felicità in questi duri anni accademici, per me è importante e doveroso provare ad esprimere riconoscenza verso coloro che hanno reso possibile il mio arrivo al traguardo, attraverso esse.

Il primo ringrazimento, il più importante, va alla mia Famiglia. Mamma, Papà, Sorella e Nonna che in questi anni mi hanno donato molto. Pazienza quando c'erano momenti che necessitavano di attesa, forza quando a questi bisognava reaggire, intelligenza nel saper distinguere i due, fiducia anche quando ero io stesso a non darmene, coraggio a non mollare sono stati per me punti chiave per andare avanti. Il sostegno morale ed economico, alle quali inizialmente, forse ingenuamente, davo poco peso, è divenuto una solida certezza nei momenti più difficili, gli ultimi.
Ragion per cui vi dico *Grazie*, di vero cuore.

Ringrazio i miei amici, quelli storici, con i quali ho vissuto e vivo momenti indimenticabili ogni volta che si presenta l'occasione. Anche se ci sentimo troppo poco e ci vediamo meno, ciò che ho imparato da loro lo porterò con me per sempre. Oggi sono questo anche grazie a loro.

Nell'esprimere gratitudine non posso fare a meno che ringraziare chi nei primi anni di questo percorso, trascorso nel residance universitario *Villa Claretta*, mi è stato vicino donandomi gioia, attimi di svago e serate passate nella hall per non sentirsi meno soli, anche se lontanissimi da *Casa*. Ringrazio quindi Alessandro, Lisa, Davide, Giovanni, Lorenzo, Alessia, Pasquale, Floriana, Francesca, Claudia e chi a partire da quegli anni mi ha fatto sorridere facendo sentire meno il peso universitario.

Ringrazio Angelo, Salvo, Marco, Zak, Giovanni e Francesco, colleghi ed amici che durante gli ultimi anni sono stati parte indispensabile della mia vita, sia all'interno del mondo universitario che fuori. Conciliare così bene studio e svago senza di voi sarebbe stato impossibile. *Grazie*.

Ringrazio il mio relatore Marcello Chiaberge e tutto il PIC4Ser. Simone, Gianluca, Angelo, Jurgen, Lorenzo, Luca, Vittorio, ed altri ragazzi del dipartimento che durante i mesi di sviluppo hanno sempre mostrato disponibilità ogni volta che chiedevo aiuto.

Contents

List of Figures

Chapter 1

Introduction

From the first days of the aviation, aircrafts required the continuous attention of a pilot to fly safely. Because the growing flight duration and in order to simplify some tasks, helps pilots during the filgth, for security and other important reasons, was introduced automatic pilot system.
Initially developed in the military field, and subsequently introduced in the civil field, nowadays autopilot systems are mandatory for most air vehicles except those of small-size (less than 20 seats).
An autopilot is a system used to control the trajectory of an aircraft without constant 'hands-on' control by a human operator being required.

In the last years, along with the development of increasingly sophisticated, safe and precise autopilots, we have seen a growth of UAVs (Unmanned Aerial Vehicles). In the past, these technologies was associated only to a military goals, but today, thanks to the reasonable price and an spread of autopilot drones firmwares (a software built ad-hoc for specific hardware) developed by different communities, these technologies have become more commons for the hobbyist or professionals peoples.
This type of system can be applied to all motor vehicles like boats, cars, aircrafts, rovers etc. One of the most important areas of application is the aviation, and in particular that of UAVs.

An UAV is *"A powered, aerial vehicle that does not carry a human operator, uses aerodynamic forces to provide vehicle lift, can fly autonomously or be piloted remotely, can be expendable or recoverable, and can carry a lethal or nonlethal payload."* [1].
These types of vehicles can have different designs, depending on the purpose for which they were built. One of the category of UAVs is a multirotor, a drone with one or more than one engines, which have the possibility of vertical takeoff and landing (VTOL).

Due to the great progress of technology, UAVs and autopilot are evolving up to autonomous systems and nowadays are capable of running by themselves with a defined flight mission without interventions from humans.
In general, an autopilot works in two ways: jointly pilots in order to support them during the fligth, or in complete autonomy in order to do some tasks, taking advantage of the sensor and GPS system. Because of that, this system is in charge to keep stable attitude and altitude of the drone, execute some classic maneuvers like roll, pitch and yaw, up to more complex maneuvers like flips.

One of the main objectives set from companies that develops these technologies is to minimize the architecture, developing autopilots more performant and, obviously, reducing costs.
In recent years, thanks to the growth of computational calculation of microprocessors, this goal is always more easily reachable, employing integrated and smaller systems.
So, a possible idea is to use a high-end smartphone with all of the necessary sensors integrated.

The project of this thesis tries to use the smartphone as autopilot, exploiting all its sensors, developing an app in charge to performs all necessary calculations and communicates the require power to the engines, in order to drive a drone.

1.1 State of Art

1.1.1 UAV

During the wartime, many inventions were made. Among all, the aviation sector has undergone a strong evolution with the birth of UAVs.
The first UAV was a balloon which carried explosive and dropped its payload after time-delay. They was tested during the American Civil War (1861-1865), but due to difficults to establish the exact time-delay, wind speed and weather this invention was failure.
In years of World War I, many tests about radio-controlled unmanned aircraft were carried out, but they did not pass the testing phase before the end of the war.
In the 1930s, the British Royal Navy used a primivite redio-controlled UAV, Queen Bee as aerial target practice.
During the World War II the nazis developed a more powerful UAV that could carry 907kg of exsplosive and could reach a speed of 800 km/h called Revenge Weapon 1.
After that more other experiments was made and in the 1970s and 80s Israel develop smaller UAVs which could transmitt live video with 360-degree view.
Although the UAVs technologies had a big growth during the 20th century, its first most important success was in the 1990s thanks to Predator, an UAV made by the US Department of Defense.

MAVs

A micro aerial vehicle (MAV) is a subclass of UAVs with restricted size. Modern vehicles of this type can be fly in autonomous way and, although new technologies promises to reach the insect-sized in the next future, at today can reach dimensions of 5 cm.
Initially build to support militaries in war zones, like *"Black Hortnet Nano"* Figure 1.1, an helicopter unmanned aerial vehicle developd by British Army in 2012 with dimensions of 10cm x 2.5cm and equipped with a camera which helps soldiers to discovers any hidden dangers, today are used to other scope like aerial photography or in context of aerial robotics.

Figure 1.1: Black Hornet Nano Helicopter UAV.

1.1.2 Autopilot

The autopilot is the most important components in an UAV. It is a software which can be more or less complex and is used to drive everything which is supposed to be unmanned, like small private boats, automobiles, bike, or also submarines, aircraft or oil tankers.
The first use of an autopilot was a airplane and ship stabilizer in 1914 by Elmer Sperry and his company *"Sperry Gyroscope Company"* [2].
Sperry and Nicolas Minorsky works togheter and made a huge contribution to the autopilots, formulating the *"Proportional-Integral-Derivative"* control law. Today autopilots are used in simple operations such as to keep heading or altitude, or in more complex operations like turning or keeping control of unstable vessels.

1.1.3 Guidance

The guidance system of an autopilot is a very important part, which determines the path of travel of the vehicle's current location to a designated target, based on commanded signals given by an operators, such as altitude speed, roll, pithc and yaw angel etc. This system is usually part of a GNC system (Guidance, Navigation and Control), and in general computes the instructions for the control system which comprises the attuators (e.g. thrusters), which are able to manipulate the flight path and the orientation of the vehicle without direct human control.
The guidance system has three sub-sections: Input, Processing, Output. The Input section include, principally, sensors data and also other information sources. The Processing section integrates this data and determine what actions are necessary execute. Then this is give to the outputs which can directly affect the system's course. The outputs may control speed by interacting with devices like turbines, or they may more directly alter course by actuating, for examples with rudders.

1.1.4 Control

The control system refers to the forces manipulation needed to execute guidance commands and maintaining vehicle stability.
There are two main division in control theory, namely, classical and modern. The classic control theory is limited to single-input and single-output (SISO) system design. This consists in a simple single variable control system with one input and one output. The modern control theory is carried out in the state space and can deal multiple-input and multipe-output (MIMO) systems.
In general SISO system is less complex than MIMO. For this reason in the industrial applications it is preferred use the classical control theory rather than the modern control theory.
The most common controllers design using classical control theory is *"PID controllers"*.

1.2 Motivations

The reasons about use an UAV or an MAV are various.
In security and military field, this type of technology has always been used, although over the years it has expanded the purpose of use. At the beginning UAVs were used only in war missions. Nowadays instead it are exploited in missions DDD (Dangerous, Dirty and Dull Rules) in order to monitor highly protected areas with UAVs Stealth which are not traceable by radar, or environmental monitoring after nuclear or chemical contamination or in dull missions as monitoring for many hours with much less effort by the soldaries.
Other type of use in military or secutiry sector are:

- Intelligence, Surveillance and Recognition (ISR)
- Suppression of Enemy Air Defences (SEAD)
- Military training
- War on terrorism
- Decoy for missiles
- Provision

Civil and scientific fields are also two important areas where this technologies are used.
Due to the growth of sensor technologies and thanks to the development of small integrated systems, like digital cameras or advanced sensors like Laser Imaging Detection and Ranging / Light Detection and Ranging (LiDAR), these areas have adopted the UAV/MAV technologies.
In Remote Sensing, that is scientific-sector which has the goal to retrieve qualitative and quantitative informations about environment and objects through a electromagnetic sensor that interacts with the specific surface, MAVs are used to create agricultural crops maps, monitor the health of vegetation, create maps after natural disasters or mapping of thermal dispersions of buildings. In agriculture they are used to irrigate fields with toxic fertilizers or figth deforestation. Another field of use is the expansion of the internet connection, using these drones in order to decrease the digital division of

the planet, which remains a sector of interest by some big company like Google and Facebook.
MAVs and UAVs are used also:

- Weather
- Microbiology
- humanitarian aid
- Survaillance
- Parcel delivery
- Hobby and recreational use
- Journalism

Though these types of vechicles can be use to execute different missions, almost all the components that it have installed onboard, are the same.
In general, to make an UAV/MAV fly we need to some physical components like actuators, frame, payload, power supply system and, in order to control it, need of other components like RC, Ground Control Station (GCS), sensors, GPS system, camera and obviously a software that is the smart component which make the decisions to move the drone.
Thanks to technological progress it has been possible to make some of these components smaller and smaller. Nowadays some devices like 'high-end' smartphones have almost all these components integrated and the idea to use a software/application installed on it, is more and more current.
The mainly reason to use a smartphone and an application as autopilot is about economic, in fact some of these components have high price if bought individually. At today with fews hundred of euro we can buy an integrated system with all onboard.
Other motivations to replace some system with an smartphone are:

- Small size dimensions
- Possible use in different types of unmanned vehicles
- Easy implementation
- Possible to extend some basic functionalities with other more complex
- Higher performance

1.3 Objectives

1.3.1 Main Objective

Develop and implement an application for Android high performance smartphones which play autopilot role for an UAV type Quadrotor.

1.3.2 Specific Objective

- Create a C firmware for arduino in order to exchange data coming from Radio-Commands (RC) to Smartphone, read the data coming from the Smartphone and generate PWM (Pulse-Width Modulation) for engines.
- Implement threads which generate the needed power to the engines, to execute Vertical Take-Off and Landing (VTOL), keep the attitude and altitude of the drone and to perform some maneuvers coming from RC.
- Establish a connection between the smartphone and a Ground Control Station (GCS) to communicate telemetry information and other information about the UAV.
- Establish another connection for video stream transmission
- Implement a thread which read commands from GCS to take some photos.

Chapter 2

The History of UAVs

Man's greatest dream has always been to fly. It has very ancient roots and many have been the myths and legends belonging to the most varied cultural traditions that follow this theme.
It is not easy to identify which was the first UAV (literally) built in history, but initially they had little or no control by the man, following a ballistic trajectory.
The idea of building a *"flying machines"*. was conceived about 2500 years ago, in ancient Greece and in China. The first example of an autonomous flying machine, which still follows a ballistic trajectory, is credited to Archita by the city of Tarantas (or Tarentum), known as *"Archita il Tarantino"*.
Probably he was the first engineer, designer and implementer of various mechanisms. In 425 a.c. he built a mechanical bird, called 'the pigeon' Figure **??**. According to Cornelius Gellio in his Noctes Atticae, the bird was made of wood, well balanced with weights, and it flew using air (most likely steam) closed in its stomach to precipitate after about 200 meters of flight.

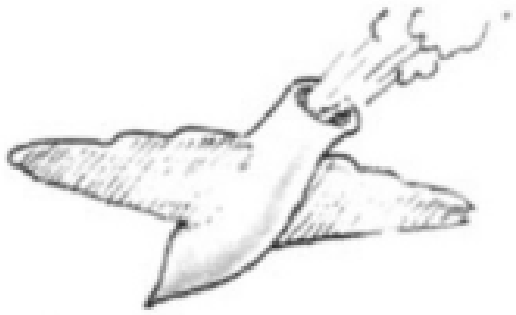


FIGURE 2.1: "Il Piccione".

In the Renaissance, however, there was a strong interest from the western world towards the theme of flight, thanks to the studies followed by Leonardo da Vinci who, in 1483, designed the aerial screw (or air gyroscope), a plane able to fly. He realized that an airplane could have been flying with a simple spiral that had to be maintained in a circular and continuous motion, to guarantee flight by exploiting the density of the air.
About 20 years earlier, however, in Japan, some testimonials say that a similar object was invented. A toy composed of a small propeller connected to a small stick that allows it to get up in flight after quickly rubbing the barrel between the palms of the hands, or thanks to a rope. It is called *"Take-tombo"* Figure 2.2.

It was then in the seventeenth century that, with a new methodological awareness, other scholars devoted themselves to the aviation sector and in 1783 the important result of the first flight of a balloon was achieved.

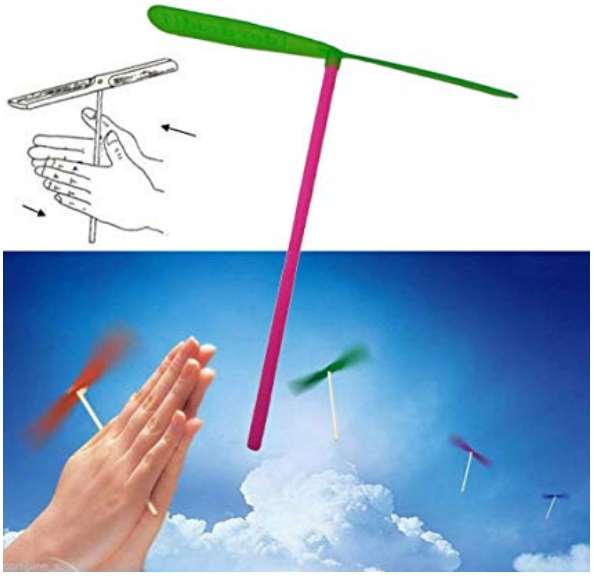


FIGURE 2.2: Taketombo.

2.1 History of first UAV

Austrian warfighting

The first evidence of the use of a UAV in history dates back to July 1849, when Venice has been besieged by Austrian forces. They tried to throw around 200 incendiary balloons carrying bombs that had to be dropped from the ball with a fuse of time over the besieged city.
At least one bomb fell in the city; however, due to the change of wind after the launch, most of the balloons missed the target and some came back on the Austrian lines.

World War I

Other prototypes of unmanned aircraft appear during the First World War: the first sample is *"Aerial Target"* Figure 2.3 used in 1916 controlled through a radio command techniques.

FIGURE 2.3: Aerial Target Sopwith.

On 12 September of the same year, the *"Hewitt-Sperry"* Figure 2.4 automatic plane made the first flight showing everyone the concept of an unmanned aircraft, that was driven thanks to various gyroscopes installed internally.

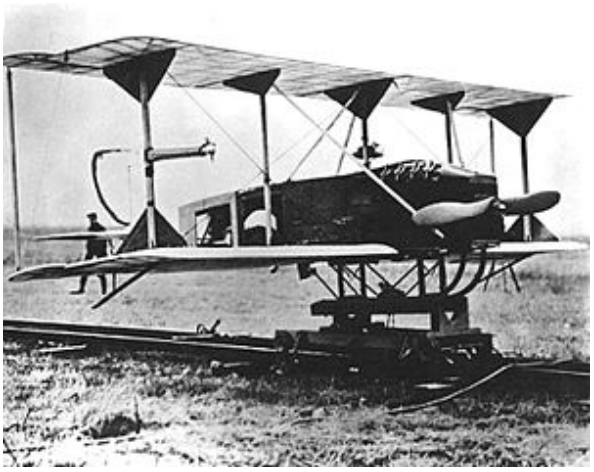


FIGURE 2.4: Hewitt-Sperry Aircraft.

Interwar period

During the time periods between World War I and World War II, technological development allowed some companies and military sectors carry on project in order to convert some aircraft models into a remote controlled aircrafts.
Three *"Standard E-1"*s Figure 2.5 were converted in drones.

FIGURE 2.5: Standard E-1 Aircraft.

The *"Larynx"* Figure 2.6 was an early cruise missile in the form of a small monoplane aircraft that could be launched from a warship and flown under autopilot.

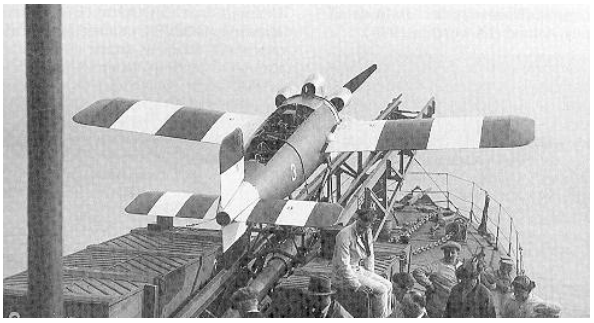


FIGURE 2.6: Larynx Aircraft.

In 1931, the British developed the *"Fairey IIIF floatplane"* Figure 2.7, building a small batch of three, and in 1935 followed up this experiment by producing larger number of another radio-controlled target, *"DH.82B Queen Bee"* Figure 2.8.

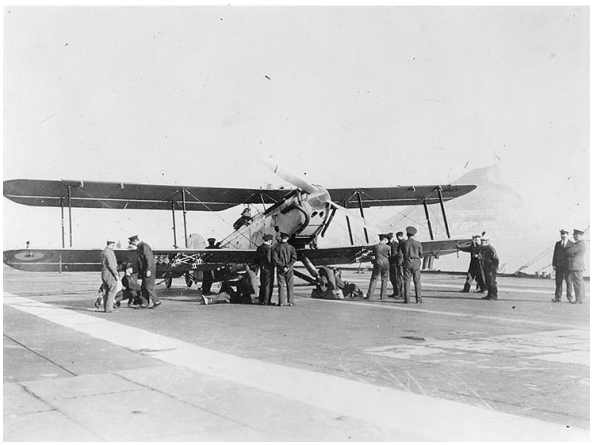


FIGURE 2.7: Fairey-IIIF Aircraft.

FIGURE 2.8: QueenBee Aircraft.

It is said that the name of *"Queen Bee"* has led to the use of the term *"drone"* for unmanned aircraft, expecially when they are radio-controlled.

World War II

The first large-scale production was by Reginald Denny. During the World War I he served the British Royal Flying Corps and after he emigrated in United States where, thaks his interest in radio control model aircraft, in the 1930s formed *"Reginald Denny Industries"*. He belived that low-cost radio control aircraft was useful for training anti-aircraft gunners, and in 1935 he demonstrated to US Army a prototype target drone, the RP-1. After more than one evolution of RP-1, in 1940 Denny and his partners won an Army contract for the radio controlled RP-4, which called *"Radioplane OQ-2"* Figure 2.9.
About 15.000 were built for the army during World War II.
Also the US Navy before and during the WW II carry out some experiments, and the result reached has been the *"Curtiss N2C-2"* Figure 2.10, a drone that was controlled from another aircraft called TG-2. After, the USAAF (US Army Air Forces), adopted this concept and, altering some obsolescent aircrafts, they could use thousands of target drones which were remotely radio-controlled.

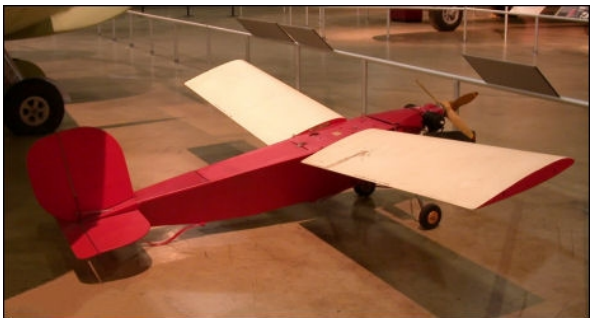


FIGURE 2.9: Radioplane OQ-2 Aircraft.

FIGURE 2.10: Curtiss N2C-2 Aircraft.

Cold War and Reconnaissance Missions

In the post-WW II, Radioplane followed up the success of OQ-2 target drone. The US military obtain a number of similiar drones and the Globe company built a series of targets which evolved through the *"KDG2"* and *"KDG5"* Figure 2.11 pulsedjet-powered.

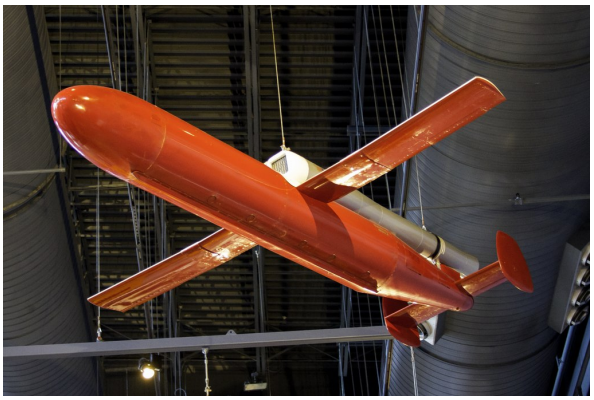


FIGURE 2.11: KDG5 Aircraft.

The success of drones as targets led to their use for other missions and the *"Ryan Firebee"* Figure 2.12 was used for some reconnaissance missions.
The US developed other more specialized reconnaissance drones, and the most important was *"Lockheed D-21"* Figure 2.12 which had maximum speed in excess of Mach 3.3 and an operational altitude of 90,000 feet.

(A) Ryan Firebee.

(B) Lockheed D-21.

FIGURE 2.12: Two representations of target drone from the 1951 (A) to 1971 (B)

2.2 Different type of UAVs

Una prima classificazione degli UAVs può essere fatta analizzando lo scopo per il quale sono stati costruiti. In generale un UAV rientra all'interno di una di queste sei categorie:

- Civil and commercial UAVs
- Combat
- Logistics
- Reconnaissance
- Research and development
- Target and decoy

Vehicles can be categorised also in terms of range/altitude or according to aircraft weight.

2.2.1 Range/Altitude classification

In this type of category we can find two main subclass:

MALE

Medium Altitude, Long Endurance. This types of vehicles can flies at an altitude window of 10,000 to 30,000 feet (3,000—9,000 m) for extended durations of time, typically 24 to 48 hours.
In this class belong:

- Hand-held 2,000 ft (600 m) altitude, about 2 km range
- Close 5,000 ft (1,500 m) altitude, up to 10 km range
- NATO type 10,000 ft (3,000 m) altitude, up to 50 km range
- Tactical 18,000 ft (5,500 m) altitude, about 160 km range

HALE

High-Altitude, Long Endurance. This types of vehicles have functions optimally at high-altitude, about 60,000 feet, and are capable of flights for considerable periods of time without recourse to landing. In this class belong:

- Hypersonic high-speed, range over 200 km and supersonic (Mach 1–5) or hypersonic (Mach 5+) 50,000 ft (15,200 m) or suborbital altitude

- Orbital low earth orbit (Mach 25+)
- CIS Lunar Earth-Moon transfer

2.2.2 Aircraft weight classification

In this category have three class:

- Heavier UAVs
- SUAS (Small Unmanned Aircraft System) - approximately less than 25 kg
- MAVs (Micro Air Vehicles) - the smallest UAVs that can weigh less than 1g

Chapter 3

Autopilot

An autopilot is a mechanical/electronic device that can drive a vehicle without the help of a human being. Most people associate autopilot specifically with airplanes, but autopilots for boats and ships are called the same way and serve the same purpose.

The autopilots for the airplanes have greatly simplified the flight. Even the most stable airplanes require constant attention from the pilot in order to fly. In the early days of air transport, the attention required to the crew was very high and the effort was high.
Modern autopilots are actually software that runs on a computer that controls the plane. This software reads the position and the current orientation of the airplane from an inertial navigation system, and controls a fly-by-wire system that drives it.

However, such a system accumulates errors over time. Errors are corrected using a satellite navigation system and altimeters. The discrepancies between the two systems are solved by digital signal processing, often a 6-dimensional Kalman filter (rolling angle, yaw angle, altitude, latitude and longitude).

Proper operation of autopilots is essential for aircraft safety. They are therefore designed to cope with failures, programming errors and any other eventuality without dropping the aircraft.
The airliners are generally equipped with at least one redundancy of two autopilots, in the case the autopilot used has a fault, the computer be present by inserting the second automatism and signaling the anomaly to the crew; this type of redundancy is called "fail-operational".

The special operating system that runs on the processor provides a virtual machine of itself. The virtual control software runs on this virtual machine. Therefore, the software never interfaces directly with the aircraft control electronics, but acts instead on a software simulation of the latter.
In this way any software error, which tends to produce gross and incorrect errors clearly, can be detected and discarded.

3.1 Born of first autopilot

The history of the autopilots starts when Lawrence Sperry, a young boy coming from a family of inventors, was invited to the Concours de la Securité en Aéroplane (Airplane Safety Competition) near the Seine River on June 18, 1914. Using the idea of the gyrocompass invented by his father, Elmer A. Sperry, Lawrence repackaged it to weigh less than 40 lbs, which he called the gyroscopic stabilizer.

Installed the autopilot on his single engine Curtiss C-2 biplane, Lawrence and his French mechanic Emil Cachin set off to demonstrate his safety improvement project to a careful and unbelieving crowd. Lawrence and Cachin began to fly over the crowd.
On the first pass, Lawrence removed his safety harness and flew through the crowd holding his hands over his head while the aircraft remained level and straight.

During the second pass, Cachin climbed out on the starboard wing and moved about 7 feet away from the fuselage, and Sperry's hands were still over his head. After that, Cachin moved out on the wing, and the aircraft, due to the shift of weight, momentarily vacillated, but the gyroscope-equipped stabilizer immediately took over and corrected the attitudinal change. The Curtiss C-2 biplane continued smoothly down the river, and this time the crowd was thunderstruck. As they passed the grandstand, there was Cachin on one wing and Sperry on the other, with the pilot's seat empty.
There was the plane, which flew serenely along with the pilot and the mechanic on the wings, which easily greeted the spectators.

Awarded first prize in the competition, Sperry received 50,000 francs ($10,000 USD) and became famous overnight, opening the evolution of automatic flight. [3]

Since then, many researches and as many developments have been made by mathematicians and engineers, and nowadays several laws have been formulated for the control of unmanned vehicles. Some of them are most complex than other, and the correct use of them depends of the dynamic model of the system. In mathematics, a dynamical system is *"A system in which a function describes the time dependence of a point in a geometrical space"*. [4].
In Control engineering the attention is focused on the modelling of a diverse range of dynamic systems, and design the controllers that will cause these systems to behave in the desired manner.
This discipline has a wide range of application from the cruise control present in modern cars to the flight and propulsion system of the commercial airlines. To implement control systems, electrical circuits, digital signal processors and microcontrollers can be used.

3.2 Control Loop

In general, control engineering use feedback when projects the control system.
Feedback occurs when the output of a specific system is routed back as input as part of cause-and-effect chain that forms a loop.
The control loop is the main building block of a control systems, and consists in control functions and physically components necessary to adjust the value of a measured process variable (PV) to match the value of a desired setpoint (SP).
Control loop includes the controller function, the process sensor and the final control element (FCE).
The control loop feedback mechanism widely used in industrial control systems is the *"PID controller"*.
This type of controller implements the following mathematical formula:

$$u(t) = K_p e(t) + K_i \int_0^t e(t')dt' + K_d \frac{de(t)}{dt}$$

Where the terms have the following meaning:

- K_p is the coefficient for the proportional term.
- K_i is the coefficient for the integral term.
- K_d is the coefficient for the derivative term.
- All this coefficient are non-negative.
- Can usually be initially entered knowing the type of application, but they are normally refined or tuned.

K_i and K_d in the *"standard form"* are replaced by $\frac{K_p}{T_i}$ and $K_p T_d$.
The advantage of this being that T_i and T_d have some understandable physical meaning, as they represent the integration time and the derivative time respectively.

3.2.1 Mains loops

In general UAVs use two main loops or combination of the two.

- **Open loop:** This loop provide a control signal (right, left, up, down) without feedback from sensor data.
- **Closed loop:** This loop provide a sensor feedback in order to adjust the behavior such as stability.

3.3 PID Controller

A Proportional-Integral-Derivative controller is used in a variety of the applications requiring continuous modulated control. It calculates an *"error value e(t)"* as a different between the desired setpoint and the misured process variable, and add or remove the corrections based on proportional, integral and derivative terms.
The first analysis and practical application of this control law was in automatic steering systems for ships. After that, also in the manufacturing industry was applied, implemented first in pneumatic and than in electronics controller.
Today is used in all applications which requires accurate and optimised automatic control.
The diagram below shows the principles of how proportional, integral and derivative terms are generated and applied. The controller continously calculates an error value *"e(t)"* as differance between a measured process variable *"PV = y(t)"* and a desired setpoint *"SP = r(t)"* and applies the correction based on Proportional Integral and Derivative terms.
It tries to reduce the error over the time by adjusting a control variable *"u(t)"*.

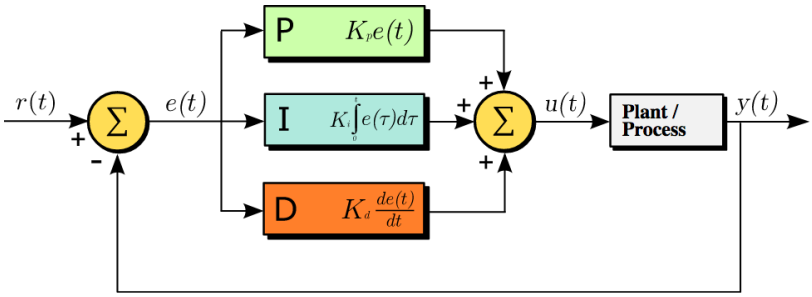


FIGURE 3.1: A block diagram of a PID controller in a feedback loop. *"r(t)"* is the setpoint (SP) and *"y(t)"* is the measured process value (PV)

In this model terms have the following meaning:

- **Term P:** This term is proportional to the current value $SP - PV = e(t)$. If the error is large and positive, the output will be proportionately large and positive, considering the K gain factor. Using proportional control alone in a compensated process, will cause an error between the setpoint and the actual process value, as it requires an error to generate the proportional response. Without an error there is no corrective response.
- **Term I:** This term takes into account the past values of $SP - PV = e(t)$ error and integrates them over time to produce the term **I**. After the application of proportional control, if there is a residual error, the integral term **I** tries to eliminate the residual error by adding a control effect due to historical cumulative value of the error and, when the error is removed, the integral term will cease to grow. This will result in a reduction in the proportional effect when the error decreases, but this is offset by the increase in the integral effect.

- **Term D:** Sometimes is called *"anticipatory control"*. This term is an estimation of the future trend of the $SP - PV = e(t)$ error, based on its current rate of change. It is effectively trying to reduce the effect of the residual error, exercising the influence of the control generated by the change in the error rate. The effect of control or damping is greater if there are more rapid changes.

Though the PID controller has three control terms some applications prefers to use only one or two in order to produce an appropriate control. Based on which parameters is used the controller is called PI, PD, P or I.
The PI controller is common due to the fact that the derivative term is so sensitive to noise, and without the integral terms the system may not reaches its target value.
However the PID algorithm does not guarantee the optimal control of the system stability. When a system does not reaches the desired stability we can choose others control laws which can be more complex to implement.
Figure 3.2 shows the response which change according to different values of Kp.

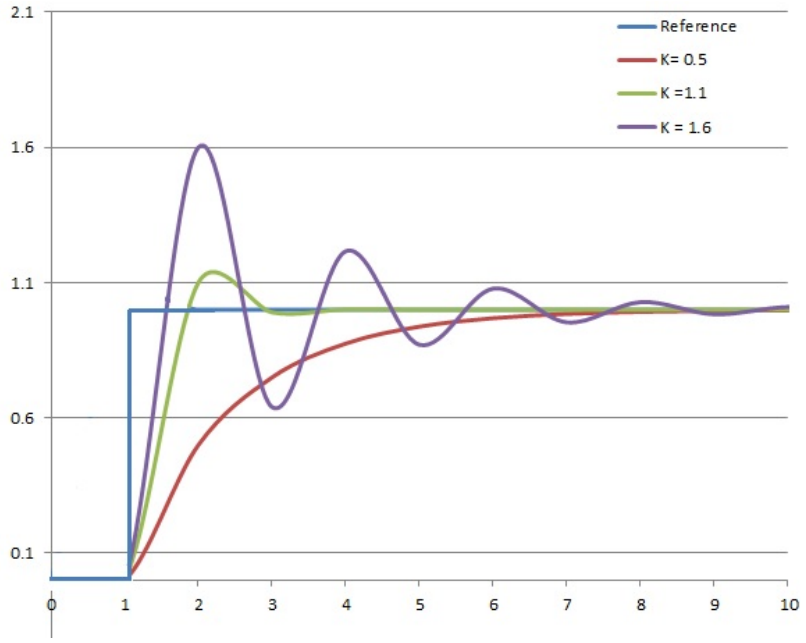


FIGURE 3.2: Response of PV to step change of SP vs time, for three values of Kp (Ki and Kd held constant) [5]

In order to improve the control system performance we can merge the feedback control (closed-loop) of the PID controller with feed-forward control (open-loop). The feed-forward value alone can provide the larger slice of the controller output.
The PID loop in this situation uses the feedback information to change the combined output to reduce the remaining difference between the process setpoint and the feedback value.

3.3.1 Loop Tuning and Stability

In order to obtain the desired control response is necessary to tune the control parameters (proportional, integral and derivative gain). Stability and no unbounded oscillation is a requirement and different systems have different behavior.
Although there are only three parameters, this type of regulation for PID controller is a difficult problem. There are various methods in order to loop tuning and more sophisticated of them are the subject of patents.
Often the default configuration of the parameters is acceptable, but performance can be improved by careful tuning. Usually at the start it is adjust repeatedly through computer simulations until the closed-loop system performs.

When the PID controller parameters are incorrectly the input can be unstable, its output can oscillate more or less intensely and diverges. This cause instalibilty of the system, in general due to the excess gain mostly when have a significant lag.
The origin of instability can be seen in mathematical formula in Laplace domanin.
The loop transfer function is:

$$H(s) = \frac{K(s)G(s)}{1 + K(s)G(s)}$$

with

- $K(s)$ PID transfer function
- $G(s)$ is the system transfer function

When the closed-loop diverges for some s the system is called unstable.
This happens when $K(s)G(s) = -1$. When $|K(s)G(s)| = 1$ have a phase shift. Instead when $K(s)G(s) < 1$ the stability is guaranteed.

3.3.2 Manual Tuning

When you need to set up system optimization manually, one of the methods is initially to set K_i and K_d to zero. After increase K_p until the output oscillates, and generally holds this value. If we want a quarter amplitude decay we must set the K_p value about half of that value.
Then increase K_i until the offset is corrected in right time for the process, but take care because too much K_i cause instability.
Finally, increase K_d, if necessary, until the loop is ready to reach its reference after a load disturbance. Again, too much K_d cause an excessive response and overshoot.

Parameter	Rise time	Overshoot	Settling time	Steady-state error	Stability
K_p	Decrease	Increase	Small change	Decrease	Degrade
K_i	Decrease	Increase	Increase	Eliminate	Degrade
K_d	Minor change	Decrease	Decrease	No effect in theory	Improve if K_d small

FIGURE 3.3: Effects of increasing a parameter independently

3.3.3 Tuning PID by software

Modern industries does not tune the PID coefficients in the tune loops using the manual calculation methods. They using PID tuning and loop optimization software to obtain good results. These software retrieve the data, develop a process model and suggest an optimal tuning.
A self-tuning feature in implemented in some digital loops controllers, but find the optimal values is always a difficult work. Advances in automated PID loop tuning software also deliver algorithms for tuning PID Loops in a dynamic or non-steady state (NSS) scenario.

3.3.4 Limitation of PID controller

Although the PID controllers are applied to a wide range of applications, and, with an easy tuning of parameters, almost always have good results, in general not provide optimal control.

The main problem is that being a feedback control system with constant parameters, the PID controller has no direct knowledge of the process, and performance is reactive and a compromise. It is the best controller when you have not a model of the process, but for better performance need to modelling the actor of the process.
One improvement is to incorporate a feedforward control with knowledge about the system.
Other two big problems regards:

- **Linearity:** Performance in non-linear systems is variable
- **Noise in derivative:** It amplifies the measurement of the highest frequency or process noise that can cause large amounts of changes in the output

3.4 Autopilots software

The Autopilot or fligth stack is a software that runs onboard the UAV systems and provide total or partial support to execute some tasks. Being a real-time system, it need to rapid response to changing sensors data. For this reason in addition to software, it is also important to choose the correct architecture.
In the civil sector there are many open-source sotfware that taken into account, while all of these are can be tracked back to the two main projects:

- Dronecode
- ArduPilot

Other noteworthy software porject are:

- Paparazzi UAV
- LibrePilot
- Flone
- OpenDroneMap

3.4.1 Communication

The autopilot software manages also the communications part between a ground station and UAV. The most important data that exchanges the two parts are data for remote control and telemetry data of the UAV.
The first types of UAVs had bidirectional narrowband radio links that transmitted command and control data (C & C) and telemetry data on the state of aeronautical systems to the remote operator, while for military UAVs satellite links were used. To date, this type of connection is still preferred in some military missions. Was implemented also a separate analog video radio link if the video streaming was required.

At today, in the modern UAVs implementations instead of having two separate link to command, control, telemetry and video traffic, we use a broadband link to carry all data on the single radio link. Usually these broadband connections carry TCP/IP traffic that can be routed over the Internet.

There are some ways in order to issued the radio signal from the operator side:

- Remote network system like a satellite duplex data links for military powers
- Ground Control where an operator use a radio transmitter/receiver, a computer or the original meaning of a military ground control station (GCS)
- Another aircraft, uses as mobile control station

A MAVLink protocol is becoming increasingly popular to carry command and control data between ground control and the vehicle.

Chapter 4

Hardware used

In addition to the design and development of an Android application that played the role of autopilot, this thesis project aimed to build a drone with this autopilot on board.
To do this, after developing the software, the drone was assembled using the following components:

- Iris frame
- Smartphone Xiaomi MI 8
- Arduino Micro
- 4-in-1 ESC and power board
- Battery: 3-cell 11.1 V 3.3 Ah lithium polymer with XT-60 type connector
- 3DR Motor AC MS2213-12 950 KV
- Propellers: (2) 10 x 4.7 normal-rotation, (2) 10 x 4.7 reverse-rotation
- RC JR Propo XG8 DMSS
- Cavo USB-C a Micro B 2.0

4.1 Frame

Figure 4.1: Iris frame

The frame used for this project was that of Iris, modified appropriately for the purpose. It was deprived of the original autopilot (32-bit Pixhawk autopilot) and all the communication hardware, and

replaced with a smartphone and an Arduino microcontroller.
We chose this frame because it presents an excellent compromise between a professional drone and an experimental thesis project in which it is necessary to carry out numerous test phases. For example, it has the capacity to carry around 400 g of payload thanks to the motors and battery that provide the necessary thrust.

4.2 Smartphone

Figure 4.2: Xiaomi MI 8

The smartphone is the heart of the project. It plays the role of autopilot through the app developed. It implements the PID control law discussed above, receiving data from the radio control via the arduino controller, or imposing values to maintain stability in flight, and, via the serial port (usb), provides data subsequently used by arduino to control the engines.
We have chosen to use this smartphone for the following reasons:

- **Processor:** 4x 2.8 GHz Kryo 385 Gold + 4x 1.8 GHz Kryo 385 Silve
- **RAM:** 6 GB
- **Camera:** 12 Mp + 12 Mp
- **Optical Stabilization**
- **Geo Tagging**
- **LTE**
- **Wi-Fi:** 802.11 a/b/g/n/ac
- **Optical Zoom:** 2x

- **GPS:** A-GPS/GLONASS/BeiDou Dual-frequency GPS. In this smartphone the BCM47755 chipset has been implemented which reduces the precision deviation from 5 meters to 30 centimeters. [6]
- **Sensors:** Accelerometer, magnetic field, orientation, gyroscope, pressure, gravity, linear acceleration, rotation vector, geomagnetic rotation vector etc.
- **Value for money**

4.3 Arduino Micro

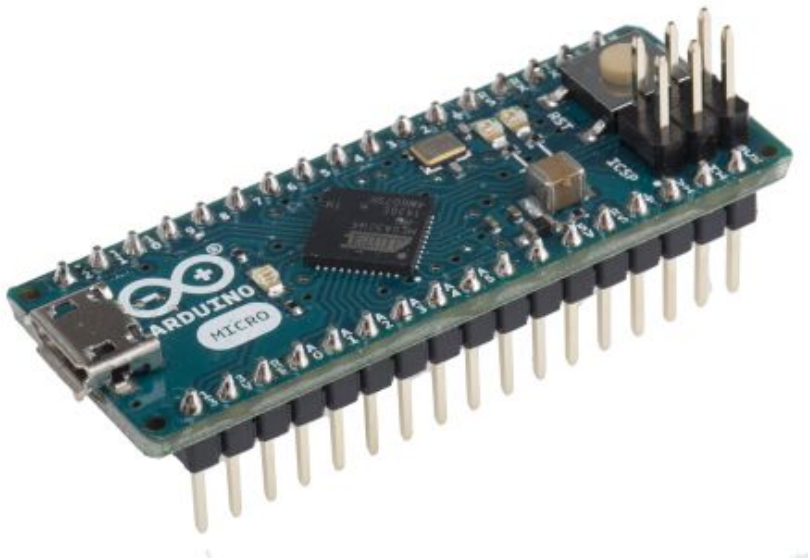


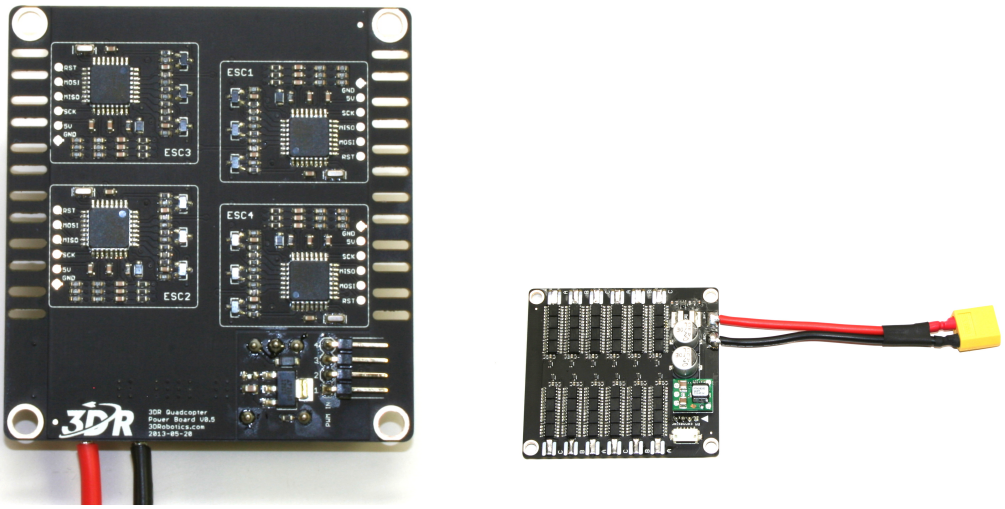
Figure 4.3: Arduino Micro

Arduino micro is a microcontroller. It is another component that has contributed to the realization of this project.
The main purpose is to manage the communication between the high level and the low architectural level of the project.
This means managing the PWMs coming from the radio control, interpreting them, assigning them a value that can be understood by the application, communicating them to the application and still reading the values coming from the application and transforming them again into PWMs.
On the other hand, these values are partly managed by the arduino software since, through a precise combination of the PWMs coming from the different channels, it is possible to arm and disarm the engines.
The reasons why you have chosen to use this type of Arduino are described below:

- Its small size
- Low cost
- Sufficient computing power
- Sufficient number of digital pins

However the use of this type of arduino also has some negative aspects, since if on the one hand it is true that the computing power is sufficient, if it is also true that the reduced dimensions allow a favorable position on board, it is also true that, the impossibility of managing in parallel all the PWMs coming from the four radio control channels causes a considerable slowdown in the main loop. The use of an arduino with the ability to manage the four channels in parallel, such as the *"Arduino Mega"*, would have improved overall performance.

4.4 ESCs



(A) Front. (B) Back

FIGURE 4.4: ESCs front/back board and power distribution

The acronym ESC for Electronic Speed Controller is a component that manages the speed of electric motors. The ESCs communicate with the Arduino control board using a digital modulation technique called *"PWM"* "pulse-width modulation". Therefore, at a high duty cycle, there is an engine that runs faster.
In this project the drone *"Iris"* ESC board has been used which includes 4 ESCs in a single board. It receives the PWM input from the Arduino microcontroller and supplies the necessary power to each motor. Each ESC is rated at 20amps capacity.
Usually the ESC receives *"PWM"* with a high logic level between 1000 µs and 2000 µs which means that at 1000 µs the motor remains stationary instead of 2000 µs turns at maximum.

4.5 Battery & flight time

FIGURE 4.5: Battery

This battery provides the necessary power required by the ESCs to drive the engines.
It has the following features:

- **Material:** Li-Polymer
- **Cells:** 3S
- **Voltage:** 11.1v

- **Capacity:** 3300mAh
- **Discharge Rate:** 35C
- **Burst Rate:** 70C
- **Voltage per cell:** 3.7V
- **Max voltage per cell:** 4.2V
- **Dimension:** 135*42*23mm
- **Weight:** 260g

4.6 Motors

FIGURE 4.6: Engines T-Motor

This is a brushless motor for airplanes and multi-rotor MS2213-12 950 kV.
The main features of the engine are:

- **Motor type:** Brushless
- **Diameter:** 27,7 mm
- **Length:** 28,5 mm
- **Weight:** 57 g
- **RPM/V:** 950 KV
- **Battery:** 2-3 S LiPo
- **Propellers:** 9x4.7-10x4.7
- **Power:** 160 W
- **Max Watts:** 220 W
- **Max Votage:** 3S

Power out put from ESC, 3S LiPo, 10x4.7 propellers:

	25%	50%	70%	100%
Amp	1.26 A	3.72 A	7.69 A	13.3 A
Wattage	18.7W	55.2W	109.2W	189.3W
Thrust	151gr	406gr	702gr	970gr

4.7 RC

The remote control used for this project is the XG-8 produced by the JR company. It consists of two components, the transmitter and the receiver.

- **Transmitter:** Is the classic radio used by the pilot on the ground.
- **Reveiver:** It is connected to the Arduino microcontroller to which it transmits the received signals.

(A) Transmitter (B) Receiver

FIGURE 4.7: RC & Receiving

The technical specifications are listed below even though the basic functions of a common radio control were used in this project.

Transmitter Specs:

- **Channels:** 8
- **Modulation:** DMSS
- **Band:** 2.4GHz
- **Receiver:** RG831B 8-channel full-range receiver
- **Programming Features:** Airplane, sailplane, helicopter
- **Model Memory:** 30
- **Mode:** Mode 1, 2, 3, 4 selectable - (mode 2 default)
- **Transmitter Battery:** 6.4V 1400mAh Li-Fe

Receiver Specs:

- **Model:** RG831B
- **Type:** Full range receiver
- **Channels:** 8
- **Modulation:** DMSS

- **Band:** 2.4GHz
- **Length:** 48mm
- **Width:** 25.5mm
- **Height:** 14.5mm
- **Weight:** 15g
- **Voltage Range:** 4.5V 8.5V
- **Antenna Length:** 150mm

4.8 Propellers

FIGURE 4.8: Propellers.

For this project the following propellers were used:

- (2) 10 x 4.7 normal-rotation
- (2) 10 x 4.7 reverse-rotation

4.9 Cavo USB-C a Micro B 2.0

FIGURE 4.9: USB-C a Micro B 2.0

This cable was used to connect the smartphone and the Arduino for data transmission.

Chapter 5

Smartphone as Autopilot

The main idea of this thesis project is the development of an Android smartphone application that performs all activity necessary to drive vehicles and communicate with a ground control station.

In this application there are three main threads that performs the necessary actions to drive the drone and communicate with a ground station.
The first thread holds drone attitude and altitude, but when receives commands coming from RC imposes this values and moves the drone in the desired direction.
The second thread collects data from the sensors, filters them and stores into variables which will be uses later by the first and third thread.
The thid thread are responsible for communicating telemetry data and other information related to the smartphone with a ground station.
Video streaming is delegated to another thread.
This application has been tested only on a quadcopter, but use on borad other vehicles, such as rover or boats, is not excluded.

5.1 Application Chart

After the main activity requires some permissions like the one for accessing the camera, microphone, location and disk, it performs the following tasks:

- Starts the thread for communication with the ground station.
- Instantiates the *"MySensors"* class object.
- Registers a listener for the battery status.
- Starts the thread to collect data from the sensors.
- If it detects the presence of a device connected to the serial port, it registers a listener to request communication. In case the permission is accepted, the object of the serial port is allocated and thread for the communication with device starts.
- A listener is registered to manage the sending of the GPS position to the ground station via *"Mavlink"* connection.
- Finally, if the permission to access the camera went well, the video streaming service starts.

In the following graph, Figure 5.1, we can see a visual representation of the work done by the application threads.

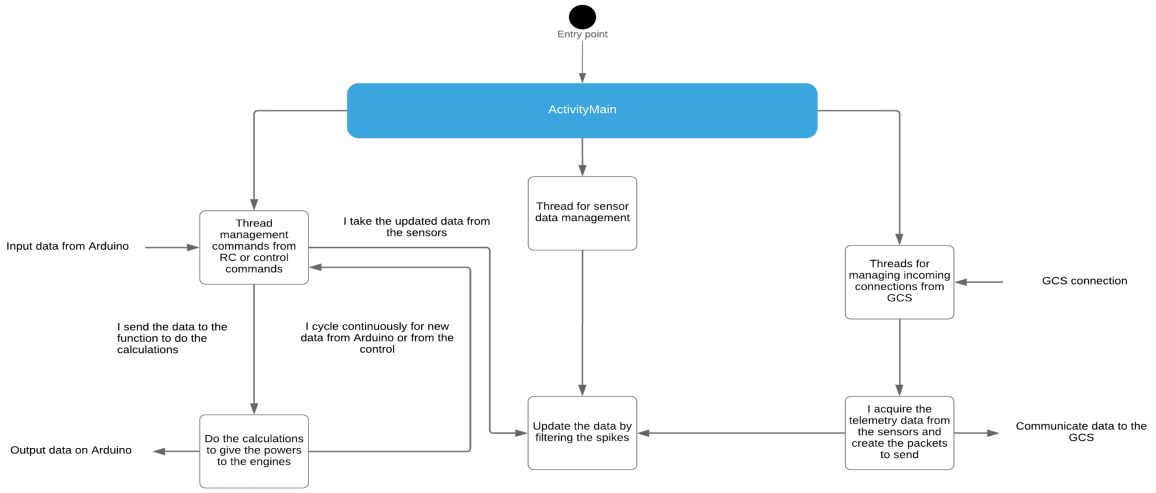


FIGURE 5.1: Schema of Application

5.2 Packages

This application is divided into two packages based on the different objective:

- *"androidAutopilot"*
- *"androidGroundControl"*

5.2.1 androidAutopilot

The first package contains all classes necessary to retrieve the information from the sensors and the RC, calculate the power to be assigned to each motor and communicates it to the Arduino microcontroller. This package is composed of five classes:

- Class *"AdkCommunication"*
- Class *"AutoPilot"*
- Class *"MySensors"*
- Class *"PidAngleRegulator"*
- Class *"PidRegulator"*

AdkCommunication

This class implements the main methods for managing the communication between Arduino and the smartphone. This happens physically through the serial port that communicates a 4 bytes buffer in which the signals coming from the RC are mapped. Each signal is a *"PWM"* and corresponds to a decimal value between 0 and 255.
A PWM (Pulse-Width Modulation) *"it is a type of digital modulation that allows to obtain a medium variable voltage based on the ratio between the duration of the positive impulse and the negative one (duty cycle). It is basically used for communication protocols where the information is encoded in the form of duration over time of each pulse."* [7].
In addition to the class constructor, which instantiates the object of the *"Autopilot"* class, the object of an internal static class *"ReceivedData"*, useful for receiving data from the RC, and receives, as a parameter, an object of type *"MySensor"*, there are four other main components such as:

- **runFeedbackThread:** This is a *"Runnable"* object that implements the work performed by a thread. In the *"run"* method the data are read from sensors, imposing angles values for roll, pitch and yaw equal to zero, and communicates them to the autopilot instance. After that the thread sleep for 100 millseconds. A.1.1

- **usbReadCallback:** This is a *"UsbReadCallback"* object of the *"UsbSerialInterface"* class. In its *"onReceivedData"* method a byte array coming from serial port are received. In this byte array we find the values that Arduino microcontroller has written, coming from the four channels of the RC and corresponding to the angles of roll, pitch, yaw and thrust. Before communicating these values to the autopilot that performs the calculations, I have disabled the *"runFeedbackThread"*. A.1.1
- **setSerialPort:** After that the *"Main Activity class"* has detected the presence of a device physically connected to the smartphone serial port, i have create an serial port object. I set some configuration about the communication and, through *"read"* method, have link the *"usbReadCallback"* to the serial port object. In this method I also started the *"runFeedbackThread"* thread. A.1.1
- **setPowers:** This method is used to communicate the byte array power to each motor from *"Autopilot object"* to Arduino microcontroller. This is done thanks to *"write"* method over serial port object. Finally i have activate again the *"runFeedbackThread"*. A.1.1

AutoPilot

This class is the core of the application. Here are assigned the powers to be given to the motors in order to reach target angles imposed by some values coming from the RC.
By combining these values with the data coming from the internal sensors of the smartphone (IMU) and applying the law of the PID controller, a bytes vector is filled and then the *"setPower"* method of the *"AdkCommunication"* class is invoked to write this array on the serial port.
There are two main methods:

- **setK:** This method sets coefficients of the PID controller law. This parameters, after a careful tuning, are used to provide the gain necessary to reach the desired angle. A.1.2
- **compute:** In this method the power for each engine is executed. This is a synchronized method, so only one thread for times can executes the calculations. It retrieves data coming from the RC as a parameter and data coming from sensors calling a method of *"MySensors"* class. After some filter about decimal precision and some transformation about the values coming from RC commands in angles degree values calling the *"getAngle"* method, the *"getInput"* method of the *"PidAngleRegulator"* class is called which provides the forces to be applied dictated by the RC commands or by the feedback thread. This forces are finally added or subtracted to each engine variable in order to obtain the desired maneuvers. At the end the byte array is filled with the variables values and it is passed to the *"AdkCommunication"* class through *"setPowers"* method which writes the array to the serial port. This method writes also a log file to debug reasons. A.1.2

MySensors

This class manages values coming from sensors. It implements the *"SensorEventListener"* interface and thaks to *"onSensorChanged"* method, which receives as a parameter an *"SensorEvent"*, performs some calculation and plot values on the corresponding graph.
Another important function is the *"resume"* method which, through the *"sensorManager"* object of the corresponding class, register the listener associated to the sensor.
In this case are register two listeners over the same method *"onSensorChanged"*, corresponding the two sensors:

- **RotationSensor:** Provide data about the attitude of the smartphone.
- **PressureSensor:** Provide data about the altitude of the smartphone.

This method performs different calculations depending on the type of sensor which has releases the event. The common objective remains that of calculating real values of roll, pitch, yaw and altitude, and to draw it in the corresponding graphs.
Although in theory you should use all the data from the sensors, some of them are subjected to excessive noise that cause *"spikes"*. To eliminate this data, we try to filter it by doing a mathematical average over a sample.

The number of data samples used in this project is 25. This allows an excellent compromise between data reliability and system responsiveness. A.1.3

PidAngleRegulator

This class provides the implementation of PID controller law. In the constructor are inizialized the law constant parameters K_p, K_i, K_d and also another parameter called *"smothingStrength"* that is used to reduce the error between the current value and the target value accumulated over time. The main method is *"getInput"* that received three float parameters:

- **targetAngle:** Is the angle that we want to reaches.
- **currentAngle:** Is the current angle.
- **dt:** Is the time difference between the previous sensor reading and the current one.

This method makes the difference between the target value and the current one and gets the main angle if the result is less than -180 or above +180 degrees by adding or subtracting 360. Next, the maximum angle at 30 degrees is blocked for safety reasons.
Then they execute the proportional, integrative and derivative part of the PID law and return a result in the form of force corresponding to the input command. For example if angles are referred to roll, the result will be the roll force. A.1.4

PidRegulator

This class is similiar to *"PidAngleRegulator"* but is used to regulates the vertical thrust force.
The only difference is that the 30 degree angle is not blocked here. A.1.5

5.2.2 androidGroundControl

The second package contains all classes required for communication with a ground station via the WiFi module.
This package is composed of four classes:

- Class *"Connection"*
- Class *"ConnectionVideoToGCS"*
- Class *"SendDataToGCS"*
- Class *"TakePhoto"*

Connection

This class manages all the necessary activities to connect the drone with a ground station following the *"MAVLink protocol"*.
It extends *"Thread"* class and so implements the *"run"* method. Almost all the work of this class is done within this method.
It instantiates an object of type *"ServerSocket"* on port 5760 and subsequently performs an infinite loop to manage incoming connections from the various GCSs. In this first phase it was preferred to have all connections managed by a single thread for simplicity.
The *"accept"* method of the *"serverSocket"* object returns *"socket"* object which uniquely identifies the connection.
Next, the input stream and the output stream of the socket is binds to a *"MavlinkConnection"* class object. It is add to a *"MavlinConnection"* list to be able to manages all connections.
After this are send the battery status and the GPS data over the connection.
The mavlink protocol provides timed and constant sending of a specific packet called Heartbeat on each connection, with a frequency of 1 Hz, in order to detect the online / offline status of the drone. For this reason, for each incoming connection, a thread is released with the purpose of managing the sending of this packet. This thread is based on the *"SendDataToGCS"* class that creates the specific

packet and handles the sending over the specific connection.
In order to handle incoming packets from each connection, another thread is dropped. It manages the initial configurations with the GCS and the reception of some packages to perform specific tasks.
Specifically, we managed the possibility of taking photos against a command given by the gcs. The protocol involves sending a specific package, called *"Mav Cmd Do Digicam Control"*, from the GCS that is managed by the application upon receipt by launching a service performed in the *"TakePhoto"* class. The picture taken is stored in the smartphone's internal memory. A.2.1
Other three methods are involved to send the information to GCS:

- **sendToAllGpsData:** This method calls *"sendGPSData"* class method, passing from time to time as parameters the different mavlink connections and an *"Location"* object type. A.2.1
- **sendToAllBatteryStatus:** This method is similar to the previous one but calls a different method, passing instead of the *"Location"* object an int type variable that indicates the battery level. A.2.1
- **sendToAllSensorsData:** This method is also similar to the previous ones. It calls againg a different method and pass an *"MySensors.SensorsData"* class object in order to send telemetry data. A.2.1

SendDataToGCS

This class performs the really data sending to the GCS.
There are four main methods that all performs the same tasks, i.e. they create the corresponding packet/packets and send it on the connection:

- **sendSensorsData:** This method creates two *"Attitude"* and *"Altitude"* packets types, and sends them in a separate thread. A.2.2
- **sendBatteryStatus:** This method create a *"SysStatus"* packet type, specifying the level battery remaining and then sends it in a separate thread. A.2.2
- **sendGpsData:** This method create a *"GpsRawInt"* packet type, specifying latitude, longitude, altitude and timestamp and then sends it in a separate thread. A.2.2
- **sendHeartbeat:** This method sends a *"Heartbeat"* packet type every 1 second. A.2.2

TakePhoto

In this class is performed the necessary activities to take a photo. It extends the *"Service"* abstract class.
When some activity calls this class the *"onStartCommand"* method start the service. It acquires the *"SurfaceView"* object from the *SessionBuilder*, creates an empty right sizes bitmap, and through the *PixelCopy* class the bitmap fills with the *SurfaceView* pixels.
After that the bitmap are saved on the smartphone. A.2.3

5.3 Libraries

In order to perform some tasks and for simplicity, some external libraries were used in this project.

- "com.github.PhilJay:MPAndroidChart:v3.0.2" : This library has been used to draw the sensors data. [8]
- "com.github.felHR85:UsbSerial:4.5.2" : This library has been used to manages the serial port to exchange data between Arduino microcontroller and the application. [9]
- "io.dronefleet.mavlink:mavlink:1.+" : This library provides a convenient use of the mavlink protocol. [10]
- "com.github.ar-android:libstreaming:1.0.0" : This library has been used to manages the video streaming. [11]

5.4 Main activity

The main activity manages the application lifecycle, instantaneously creates the objects needed for the interface and releases the threads needed for the two packages mentioned above.
This class extends *"AppCompactActivity"* and is the entry point to the application.
The Figure 5.2 show the android activity lifecycle. This methods have been overridden to provide the desired goal to the application.

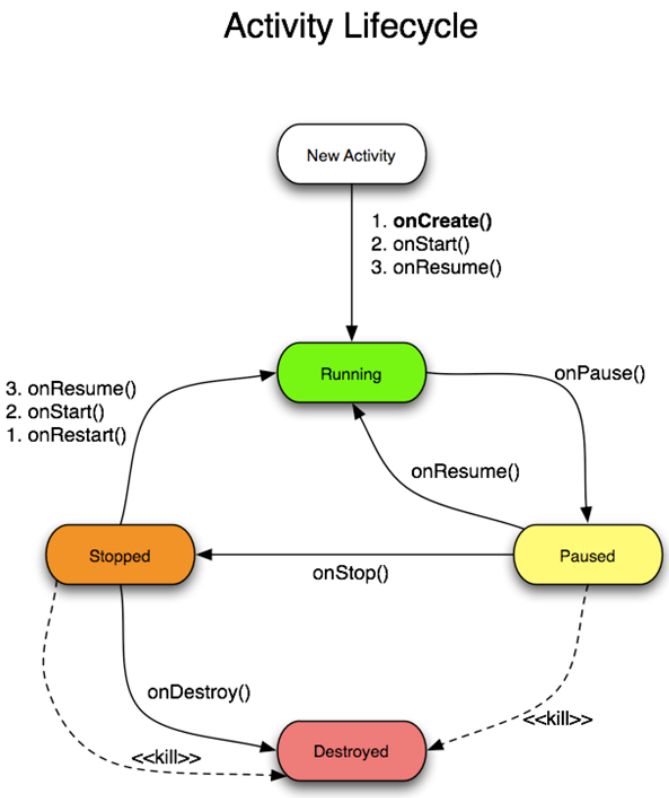


FIGURE 5.2: Activity Lifecycle.

5.4.1 onCreate

This is the first method calls when application starts. It asks permissions for:

- Camera
- Location
- Record Audio
- External Storage

After this, *"MavlinkConnection"* object, *"MySensors"* object and *"AdkCommunicator"* object are instantiated. An *"LocationManager"* object is requested to the system, and is registered a listener to the change of the battery level.

5.4.2 onStart

This method is automatically called after the *"oncreate"* method just as indicated in the application lifecycle.
Here are performed other tasks:

- The thread that manages the sensors data is started
- If an device is detected to the serial port, an usb *"BroadcastReceiver"* listener is register
- If the location permission is granted an register for location updates is performed, and the GPS data is sent over the mavlink connection created before
- If the camera permission is granted an *"SessionBuilder"* is configured and the *"RTSP server"* service is started

5.4.3 onResume

This method checks if the thread that manages sensors data is still alive and if not it will be restarted.

5.4.4 onPause/onStop

In these two methods nothing is done, because, in any case, if the activity comes in the background or if the activity comes to the fore, threads to manage sensors data, the one for communication and the one for maintaining stability drones must work.

5.4.5 onDestroy

In this method instead we have the following tasks to perform before leave the application:

- A method is called on the *"mySensors"* object in order to unregister the listener associated to each sensor.
- The serial port for communication with the Arduino is closed.
- The service that performs the video streaming is interrupted.
- Unregister a previously registered usb *"BroadcastReceiver"*.

5.5 Arduino code

The Arduino code performs all the activities necessary to put in communication the Android application and the RC to assign the necessary powers to each engine to reach a position and the desired altitude.

5.5.1 Setup

The setup() function initializes and sets the initial values.
In this function, the first instruction is *"Serial.begin(115200)"*. This instruction *"Sets the data rate in bits per second (baud) for serial data transmission."* as a definition in the official documentation of the arduino page. [12]
In this case the data rate is sets to 115200 bits per second.
Subsequently, the digital pin operation modes are defined by *"pinMode(pin, INPUT/OUTPUT)"* instruction. This instruction sets a digital pin in input or output mode. The last instruction is *"Serial.setTimeout(1)"*. This instruction *"sets the maximum milliseconds to wait for serial data when using Serial.readBytes()."*. [13]
In this case the maximum milliseconds to wait for serial data is sets to 1.

B.0.1

5.5.2 Loop

The loop() function *"does precisely what its name suggests, and loops consecutively, allowing your program to change and respond"*. [14]
This function is performed at high frequency and between the end of the execution of the last instruction inside the function, and the beginning of the execution of the first instruction in the function, in general, there is a delay of some clock cycles.
However, in this case, the loop() function executes four *"pulseIn(pin, value)"* instructions on the four channels of the RC receiver. This function *"returns the length of the pulse in microseconds"*. If *"value"* is HIGH, pulseIn() waits for the pin to go from LOW to HIGH, starts timing, then waits for the pin to go LOW and stops timing. [15]
A duty cycle is *"the fraction of one period in which a signal or system is active. A period is the time it takes for a signal to complete an on-and-off cycle"*. [16]
The radio control transmits signals with a duty cycle between 40 and 80, so transmitt signals with an active state between 1 and 2 milliseconds. Due to the fact that the RC transmits signals with a frequency of 40 Hz, in the worst case each channel receives a signal every 25 ms. Since the four instructions are sequential, this causes a minimum loop time of 25ms * 4 = 100 ms.
Subsequently, the corresponding value of each channel is mapped into a byte, assuming 0 if the *"pulseIn"* function has detected a 1 ms signal or 255 if it has detected a 2 ms signal.
At this point the values coming from the remote control are mapped and used, initially to detect the *"arm"* or *"disarm"* command, and then sent to the smartphone via the serial port.
Last task to performs is to wait for the values calculated on the smartphone, remap them in milliseconds and write the PWMs on the corresponding pins in order to gives the desired power to each engine.

B.0.2

Chapter 6

Ground control station (GCS)

A ground control station (GCS) is a control center that allows the monitoring and/or control of an Unmanned Aerial Vehicle (UAV) or "drone".
In general, when we talk about GCS we refer to the control station as hardware and software components that allows communication with UAVs through different types of links.
The most used ones are:

- Radio Links - Primary control link
- Satellite Links - Secondary control link
- Data Link

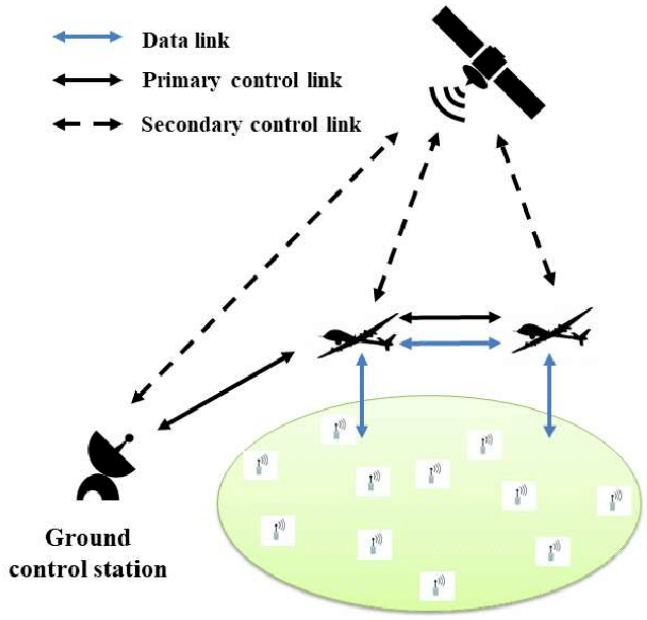


FIGURE 6.1: Basic networking architecture of UAV communications.

Generally the control link is used to give commands to the UAV in order to manage the movement and the guide. This type of connection must therefore be reliable, safe and with low delays.
The main connection is often a radio link since it has low transmission delays and can cover a fairly large area. In any case, a secondary control link is always created should the first fail or may lose contact due to the high distance between transmitter and receiver.
The second link, however, is a satellite link which, even if it has significant transmission delays with respect to the first, guarantees an almost total reachability of the area in which the UAV is located.
The last connection created between the UAV and the GCS is the data link. This link is used to transmitt all other information about the UAV, including telemetry data, GPS data, video streaming,

battery level etc.

Through these connections it is possible to transmit all the necessary data. The use of some types of links, however, is favored over others due to the reduced delays, security and higher payload.
A more precise subdivision in the class of hardware GCS can be made by differentiating those that are generally real fixed ground stations, and those that can be transported instead.

6.1 Different type of CS

The GCS hardware refers to the complete set of terrestrial hardware systems used to control the UAV. This generally includes the interface, computer, telemetry, video capture card and antennas for control connections, video and data for the UAV.

6.1.1 Portable

Figure 6.2: Portable GCS

The smaller UAVs can be operated with a traditional "twin-stick" transmitter, used for radio-controlled model aircraft. If you extend this configuration with a laptop, telemetry, video and antennas you create what is actually a ground control station.
A number of suppliers offer a combined system consisting of a modified transmitter and what appears to be a touch screen. An internal computer running the GCS software which is located behind the screen, along with video and data links.
Other large-sized portable GCSs placed inside the cases can also be found on the market. They also have a computer running the GCS software, along with video and data links. There are also large single or double screens.
Some portable GCS units are in the HOTAS (Hands on Throttle And Stick) layout which consists of a 3-axis joystick to control pitch, roll and yaw and a slide or t-bar fader can increase or decrease the UAV speed.

6.1.2 Fixed

FIGURE 6.3: Fixed GCS.

For large military UAVs you need what looks like "virtual cockpit". The pilot operator is faced with a series of screens showing the view from the UAV, a map screen and an on-board instrumentation. Control is via a traditional aeronautical joystick, possibly with Hands on Throttle and Stick (HOTAS) functionality.
The GCS consists of satellite or long-range communication links mounted on the roof or on a separate vehicle.

6.2 GCS Software

A ground station is typically a software application running on a computer, which communicates with the UAV via wireless connections. It can serve as a "virtual cockpit" and displays the real-time data coming from UAV for performance and position.
With this software it is also possible to control the UAV in flight, load new missions and monitor the parameters. Provides a map screen where the user defines the flight waypoints and sees the progress of the mission.
There is also the possibility of monitoring the streaming video from the UAV camera.
Over the years different versions of GCSs have been developed to adapt to different desktop and mobile devices.
In the field of desktop GCSs the main open-source software are:

- **Mission Planner:** Platform Windows and Mac OS X
- **APM Planner 2.0:** Platform Windows, Mac OS X and Linux
- **MAVProxy:** Platform Linux
- **QGroundControl:** Platform Windows, Mac OS X, Linux, Android and iOS
- **UgCS - Universal Ground Control Station:** Platform Windows, Mac OS X and Ubuntu

For mobile devices, the main GCS developed are:

- **Tower:** Platform Android Phones and Tablets. Open-source license.
- **MAV Pilot 1.4:** Platform iPhone and iPad. Proprietary license.
- **SidePilot:** Platform iPhone and iPad. Proprietary license.
- **AndroPilot:** Platform Android Phones and Tablets. Open-source license.
- **UgCS - Universal Ground Control Station:** Platform Windows, Mac OS X and Ubuntu

Although there is a variety of GCS software both desktop and mobile, all those with an open source license make use of the MAVLink protocol described below.

6.3 MAVLink

MAVLink (Micro Air Vehicle Link) is a communication protocol for small unmanned vehicles. MAVLink was released in early 2009 by **Lorenz Meier** with an *"LGPL license"*.
This protocol is generally used for communication between the unmanned vehicles and a ground control station (GCS). However, it can also be used for intercommunicating the systems on board the vehicle itself., for example to interface the vehicle with a *"Companion Computer"* (arduino, raspberry, linux and derivative platforms) to make "intelligent decisions" during travel.

6.3.1 MAVLink messages and data structure

A MAVLink message is a stream of bytes encoded by the sender and decoded by the recipient that the GCS sends to the vehicle's autopilot or vice versa. The message is inserted into a data structure and sent to the communication channel by adding some fields for error correction.
Each package can contain payloads or less. Those without payloads are packets with only the header, they are "acknowledgment packets" and have a minimum size of 8 bytes. All other packages containing the payload have a variable size, up to a maximum of 263 bytes.

Data structure

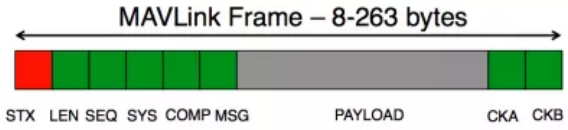


Figure 6.4: MavLink Frame.

The following table describes the bytes and their function.

Byte Index	Content	Value Range (byte)	Description
0	Sign the beginning of the package	v1.0: 0xFE	Indicates the start of a new package and the protocol version.
1	Payload length	0 – 255	Indicates the length of the message (payload).
2	Package sequence	0 – 255	Each component increases its sending sequence. Required to detect lost packets.
3	System ID	1 - 255	System identifier that sends the packet. It allows to identify different systems on the same network.
4	Device ID	0 – 255	Identifier of the device that sends the package. It allows to identify the devices on the same system, eg gimbal, IMU.
5	Message ID	0 – 255	Message identifier Defines the meaning of the payload (data) and the ways to correctly decode it.
6 to (n+6)	Payload	0 - 255 bytes	Message data, depending on the message ID.
(n+7) to (n+8)	Checksum - CRC	2 bytes	Detection of transmission errors. Hash in ITU X.25 / SAE AS-4 encoding, **excluding byte 0 of the packet start signature.**

Checksum – CRC

To ensure the integrity of the message, the *"Checksum"* is calculated for each message and inserted in the last two bytes. The CRC also has the function of ensuring that the sender and the recipient agree on the message sent.
At first the software that receives the packet checks the validity of the message exploiting the *"Checksum"*. If the packet was corrupted, it is discarded immediately.

How the protocol really works

MAVLink is nothing but a message. This is a set of data that, depending on the type, contains a constant number of bytes.
Autopilot sends and receives a stream of data coming from a physical communication channel such as socket TCP, telemetry radio, USB etc. and decodes the message in the software.
The message contains the **payload** we want to extract.
Before the payload we are interested in knowing the **Message ID** that specifies what the payload contains. The payload data is extracted from the message and inserted into a data structure based on a certain type of information.
There are different types of data structures based on the type of data we want to exchange, eg GPS data structure, command data structure etc. This structures are the same from both sender and receiver side (GCS and autopilot).
The **Message ID** allows the software to understand the type of data structure to be used for the received packet, in order to correctly interpret the data contained.

6.4 QGroundControl desktop version

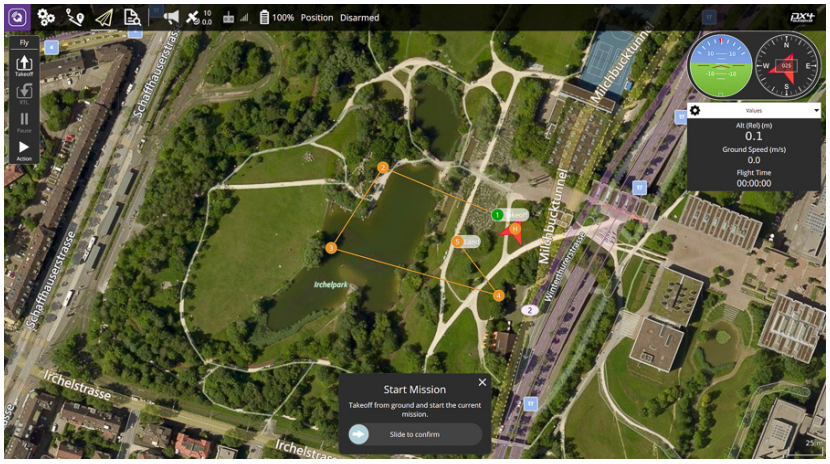


FIGURE 6.5: QGroundControl Interface

The ground control station used in this project is *QGroundControl*.
QGC is an intuitive and powerful GCS for UAVs that aims at ease of use for both professionals and beginners. It provides complete flight control and the ability to plan missions for any autopilot/drone that uses the MAVLink protocol.
This software is open-source.
The main features, as reported in the official site [17], are:

- Flight support for vehicles running PX4 and ArduPilot (or any other autopilot that communicates using the MAVLink protocol).
- Mission planning for autonomous flight.
- Full setup/configuration of ArduPilot and PX4 Pro powered vehicles.
- Flight map display showing vehicle position, flight track, waypoints and vehicle instruments.
- Support for managing multiple vehicles.
- Video streaming with instrument display overlays.
- QGC runs on Windows, OS X, Linux platforms, iOS and Android devices.

The main features that led me to choose this type of software are:

- QGC is easy to use and has an excellent user interface (UI).
- QGC is multiplatform.
- QGC is open-source.

6.4.1 Packets used

In this project, to ensure optimal management of the *MAVLink Messages* exchanged between the UAV and the GCS, it was decided to rely on an external library that guarantees the marshall and unmarshall between the data of the input packets and the classes.
This library is a Java SDK for communication which uses the Mavlink1 and Mavlink2 protocols and is called *'io.dronefleet.mavlink:mavlink:1.1.5'*.
For this project the following *MAVLink Messages* are used.

Heartbeat message

The most important message for the communication. It shows to the GCS that a system or component is present and responding.
This message is composed by following fields:

Field Name	Type	Values	Description
type	uint8_t	MAV_TYPE	Type of the system (quadrotor, helicopter, etc.). Components use the same type as their associated system.
autopilot	uint8_t	MAV_AUTOPILOT	Autopilot type / class.
base_mode	uint8_t	MAV_MODE_FLAG	System mode bitmap.
custom_mode	uint32_t		A bitfield for use for autopilot-specific flags.
system_status	uint8_t	MAV_STATE	System status flag.
mavlink_version	uint8_t_mavlink_version		MAVLink version, not writable by user, gets added by protocol because of magic data type: *uint8_t_mavlink_version*.

Only some of these fields have been used.

SysStatus message

The general system state. It shows whether the system is currently active or not and if an emergency occurred. If the system is following the MAVLink standard, the system state is mainly defined by three orthogonal states/modes:

- **System Mode:**

  Locked: Motors shut down and locked.

  Manual: System under RC control.

  Guided: System with autonomous position control, position setpoint controlled manually.

  Auto: System guided by path/waypoint planner.

- **NAV_MODE:** defined the current flight state:

  LiftOff: An open-loop maneuver.

  Landing

  Waypoints

  Vector

  Critical

  Emwegency

This message is composed by following fields:

Field Name	Type	Units	Values	Description
onboard control sensors present	uint32_t		MAV SYS STATUS SENSOR	Bitmap showing which onboard controllers and sensors are present. Value of 0: not present. Value of 1: present.
onboard control sensors enabled	uint32_t		MAV SYS STATUS SENSOR	Bitmap showing which onboard controllers and sensors are enabled: Value of 0: not enabled. Value of 1: enabled.
onboard control sensors health	uint32_t		MAV SYS STATUS SENSOR	Bitmap showing which onboard controllers and sensors are operational or have an error: Value of 0: not enabled. Value of 1: enabled.
load	uint16_t	d%		Maximum usage in percent of the mainloop time. Values: [0-1000] - should always be below 1000.
voltage_battery	uint16_t	mV		Battery voltage.
current_battery	int16_t	cA		Battery current, -1: autopilot does not measure the current.
battery_remaining	int8_t	%		Remaining battery energy, -1: autopilot estimate the remaining battery.
drop_rate_comm	uint16_t	c%		Communication drop rate, (UART, I2C, SPI, CAN), dropped packets on all links (packets that were corrupted).
errors_comm	uint16_t			Communication errors (UART, I2C, SPI, CAN), dropped packets on all links (packets that were corrupted).
errors_count1	uint16_t			Autopilot-specific errors.
errors_count2	uint16_t			Autopilot-specific errors.
errors_count3	uint16_t			Autopilot-specific errors.
errors_count4	uint16_t			Autopilot-specific errors.

Only some of these fields have been used.

Attitude message

The attitude in the aeronautical frame (right-handed, Z-down, X-front, Y-right). This message is composed by following fields:

Field Name	Type	Units	Description
time_boot_ms	uint32_t	ms	Timestamp (time since system boot).
roll	float	rad	Roll angle (-pi..+pi).
pitch	float	rad	Pitch angle (-pi..+pi).
yaw	float	rad	Yaw angle (-pi..+pi).
rollspeed	float	rad/s	Roll angular speed.
pitchspeed	float	rad/s	Pitch angular speed.
yawspeed	float	rad/s	Yaw angular speed.

Only some of these fields have been used.

Altitude message

The current system altitude. This message is composed by following fields:

Field Name	Type	Units	Description
time_usec	uint64_t	us	Timestamp (UNIX Epoch time or time since system boot).
altitude_monotonic	float	m	This altitude measure is initialized on system boot and monotonic (it is never reset, but represents the local altitude change). The only guarantee on this field is that it will never be reset and is consistent within a flight. The recommended value for this field is the uncorrected barometric altitude at boot time. This altitude will also drift and vary between flights.
altitude_amsl	float	m	This altitude measure is strictly above mean sea level and might be non-monotonic. It should be the altitude to which global altitude waypoints are compared to.
altitude_local	float	m	This is the local altitude in the local coordinate frame. It is not the altitude above home, but in reference to the coordinate origin (0, 0, 0). It is up-positive.
altitude_relative	float	m	This is the altitude above the home position. It resets on each change of the current home position.
altitude_terrain	float	m	This is the altitude above terrain. It might be fed by a terrain database or an altimeter. Values smaller than -1000 should be interpreted as unknown.
bottom_clearance	float	m	This is not the altitude, but the clear space below the system according to the fused clearance estimate. It generally should max out at the maximum range of e.g. the laser altimeter. It is generally a moving target. A negative value indicates no measurement available.

Only some of these fields have been used.

GpsRawInt message

The global position, as returned by the Global Positioning System (GPS). It is a RAW sensor value. This message is composed by following fields:

Field Name	Type	Units	Values	Description
time_usec	uint64_t	us		Timestamp (UNIX Epoch time or time since system boot).
fix_type	uint8_t		GPS FIX TYPE	GPS fix type.
lat	int32_t	degE7		Latitude.
lon	int32_t	degE7		Longitude.
alt	int32_t	mm		Altitude (MSL). Positive for up.
eph	uint16_t			GPS HDOP horizontal dilution of position (unitless).
epv	uint16_t			GPS VDOP vertical dilution of position (unitless).
vel	uint16_t	cm/s		GPS ground speed.
cog	uint16_t	cdeg		Course over ground (NOT heading, but direction of movement) in degrees * 100, 0.0..359.99 degrees.
satellites visible	uint8_t			Number of satellites visible. If unknown, set to 255.

Only some of these fields have been used.

AutopilotVersion message

As soon as the GCS connects to the autopilot, it sends a package that requires basic information on the software used by the autopilot.
The autopilot responds by sending an AutopilotVersion package which describe version and capability of autopilot software.
This message is composed by following fields:

Field Name	Type	Values	Description
capabilities	uint64_t	MAV PROTOCOL CAPABILITY	Bitmap of capabilities.
flight_sw_version	uint32_t		Firmware version number.
middleware_sw _version	int32_t		Middleware version number.
os_sw_version	int32_t		Operating system version number.
board_version	int32_t		HW/board version.
flight_custom _version	uint8_t[8]		Custom version field, commonly the first 8 bytes of the git hash. This is not an unique identifier, but should allow to identify the commit using the main version number even for very large code bases.
middleware_custom _version	uint8_t[8]		Custom version field, commonly the first 8 bytes of the git hash. This is not an unique identifier, but should allow to identify the commit using the main version number even for very large code bases.
os_custom_version	uint8_t[8]		Custom version field, commonly the first 8 bytes of the git hash. This is not an unique identifier, but should allow to identify the commit using the main version number even for very large code bases.
vendor_id	uint16_t		ID of the board vendor.
product_id visible	uint16_t		ID of the product.
uid	uint64_t		UID if provided by hardware.

CameraTrigger message

In this autopilot has been inserted the possibility to take a photo while the UAV is in flight, against a specific command received.
When the autopilot receives the command of DoDigicamControl, it takes a photo that it saves on

the device and responds to the command with the ack packet and after sends a CameraTrigger type packet.
This is a camera-IMU triggering and synchronisation message. This message is composed by following fields:

Field Name	**Type**	**Units**	**Description**
time_usec	uint64_t	us	Timestamp for image frame (UNIX Epoch time or time since system boot).
seq	uint32_t		Image frame sequence.

Chapter 7

Future Application

This thesis project was created to give the opportunity to anyone, through an application, to turn their smartphone into a real autopilot for UAVs.
Currently the implementation of the application allows, with the necessary precautions, to pilot a quadricopter through a radio control and an Arduino microcontroller, to communicate the data relating to the UAV to a control station and to perform a video streaming.
The graphical interface was not very accurate during the implementation, due to the main focus that was to guarantee a stable and precise flight, but in the immediate future we could think of improving it by selecting the type of vehicle on which we will have to install the autopilot (the smartphone), so as to configure all the parameters necessary for vehicle control.
The further expansion of this project could open a main vein in the development of autopilots for any type of vehicle, in fact the basic functions of the main types of autopilot for vehicle control do not differ much and are based on:

- A control loop that reads the data from the sensors and tries to maintain a stable attitude.
- A loop that awaits commands from one operator, one flight mission or other and gives commands to the engines to ensure the achievement of the various objectives.

In part these functions have been implemented, but clearly improvements need to be made.

Future developments could occur in the civil sector, where territorial mapping could be carried out much more easily, or in the field of emergency interventions in dangerous areas, or in the agricultural sector, where plant mapping can play a fundamental role in ensuring its health.

Therefore this project would find easy application in the technological innovation of different unmanned vehicles among which we remember:

- Rover
- Multi-copter
- Boats
- Radio-controlled model aircraft
- All vehicles that require remote or autonomous piloting

Chapter 8

Results

The main objective of this thesis was the design and development of an application for Android smartphones capable of performing the main functions of an autopilot for UAV / UGV (Unmanned Aerial Vehicle / Unmanned Ground Vehicle) systems.

During the work carried out, the foundations for hardware and software innovation in the autopilot world have been laid, even if there are several problems that are not immediately readable, such as:

1. Filter data from sensors.
2. Choice of the ideal Arduino microcontroller, Arduino with reduced dimensions offering high performance.
3. Set the data flow rate from the smartphone to the Arduino (producer / consumer problem).
4. Tuning the parameters of the PID control law.
5. Problem in reading the PWM coming from the radio control.
6. Problem related to data exchange between microcontroller and smartphone.

We have tried to solve the main problems mentioned above by adopting techniques such as:

1. In filtering the data from the sensors it was decided to implement an average of 25 samples in order to eliminate the false data that may occur, guaranteeing in any case an available data in adequate time (of the order of a few milliseconds).
2. For the choice of the Arduino it was mainly thought of the size, as the amount of calculations performed on the microcontroller was completely reduced to the detriment of the smartphone, so the achievement of the target did not require exaggerated computational powers. However, other types of microcontrollers, such as Arduino Mega, would have produced better results as they were able to manage the PWMs coming from the radio control in interrupt mode through ISR (Interrupt Service Routine), and therefore increase the frequency of the main loop.
3. In order to adjust the data rate from smartphone to arduino, in the worst case it was set in the maximum time necessary for the loop function to close a cycle. In any case, upon receipt of data from the smartphone, the microcontroller prepares a buffer so that if the commands are more than those required, only the last command is guaranteed.
4. The regulation of the parameters of the control law was carried out after several tests because the whole depends on the drone model used and therefore not deductible a priori.
5. The signals coming from the receiver of the radio control have been read by a *pulseIn* function which, although as a function is blocking, it is the only alternative to the management in interrupt of the received signals, which cannot be applied due to the limits of the chosen microcontroller.
6. Data exchange was performed through an array of bytes with a value of 0-255. Each value corresponds to a radio control receiver channel when the microcontroller supplies data to the smartphone or to an engine when the smartphone supplies data to Arduino.

These choices led to a basic version, although with some problems like:

- Stability in flight
- Difficulty in video streaming
- Interference problems when a picture is taken during video streaming

However, results such as communication between the smartphone, microcontroller and radio command, executing a PID control law on a smartphone, remotely executing commands on the drone, connecting it to a ground station for the transmission of telemetry data, GPS data and others such as Battery level, study and application of the MAVLink protocol that regulates this communication, have been achieved.

Starting from them, trying to improve what was done and expanding the work done in this thesis project, scenarios could open up in which thanks to a smartphone and an installed app, with the necessary hardware architecture required (ESCs, microcontroller, engines and battery, eliminating GPS modules, telemetry modules and streamlining the architecture), it is possible to drive vehicles remotely through an operator or even have programmed routes run independently with much more ease of realization than what is proposed for now, also reducing costs and time required for implementation.

Bibliography

[1] *Definition of UAV.* URL: http://www.thefreedictionary.com/unmanned+aerial+vehicle.

[2] *Biography of Elmer A. Sperry, 2012.* URL: http://www.ieeeghn.org/wiki/index.php/Elmer_A._Sperry.

[3] *Lawrence Sperry: Genius on Autopilot.* URL: https://www.historynet.com/lawrence-sperry-autopilot-inventor-and-aviation-innovator.htm.

[4] *Dynamical system definition.* URL: https://en.wikipedia.org/wiki/Dynamical_system.

[5] *PID Wikipedia.* URL: https://en.wikipedia.org/wiki/PID_controller#Proportional_term.

[6] *Dual-frequency GPS.* URL: https://gizchina.it/2018/05/xiaomi-mi-8-dual-gps/.

[7] *Pulse-Width modulation.* URL: https://en.wikipedia.org/wiki/Pulse-width_modulation.

[8] *Android chart view / graph view library.* URL: https://github.com/PhilJay/MPAndroidChart.

[9] *Usb serial controller for Android.* URL: https://github.com/felHR85/UsbSerial.

[10] *A Java API for MAVLink communication.* URL: https://github.com/dronefleet/mavlink.

[11] *RTSP library.* URL: https://github.com/ar-android/libstreaming.

[12] *Serial begin.* URL: https://www.arduino.cc/en/serial/begin.

[13] *Serial Timeout.* URL: https://www.arduino.cc/en/Serial/SetTimeout.

[14] *Definition of the Loop function.* URL: https://www.arduino.cc/en/Reference/Loop?setlang=it.

[15] *Definition of pulseIn function.* URL: https://www.arduino.cc/reference/en/language/functions/advanced-io/pulsein/.

[16] *Duty cycle definition.* URL: https://en.wikipedia.org/wiki/Duty_cycle.

[17] *Official site of QGroundControl.* URL: https://docs.qgroundcontrol.com/en/.

Appendix A

JavaCode

A.1 androidAutopilot Package

A.1.1 AdkCommunication

runFeedbackThread

```
private Runnable runFeedbackThread = new Runnable() {
    @Override
    public void run() {
        while(true){
            if(serialPort!=null && myTurn) {
                timePrevious = SystemClock.elapsedRealtime();
                sensorsData = mySensors.getSensorsData();
                receivedData.roll = new BigDecimal(120)
                    .setScale(0, BigDecimal.ROUND_HALF_UP).floatValue();
                receivedData.pitch = new BigDecimal(120)
                    .setScale(0, BigDecimal.ROUND_HALF_UP).floatValue();
                receivedData.yaw = new BigDecimal(sensorsData.yaw+120)
                    .setScale(0, BigDecimal.ROUND_HALF_UP).floatValue();
                receivedData.whichThread = 0;
            }
            autoPilot.compute(receivedData);
            try {  Thread.sleep(100);}
            catch (InterruptedException e) { e.printStackTrace(); }
        }
    }
};
```

usbReadCallback

```
UsbSerialInterface.UsbReadCallback usbReadCallback =
    new UsbSerialInterface.UsbReadCallback() {
        //Defining a Callback which triggers whenever data is read.
        @Override
        public synchronized void onReceivedData(byte[] arg0) {
            try {
                timePrevious = SystemClock.elapsedRealtime();
                //mySensors.setTurn(false);
                myTurn = false;
                sensorsData = mySensors.getSensorsData();
                receivedData.roll = arg0[0] & 0xFF;
                receivedData.pitch = arg0[1] & 0xFF;
                receivedData.yaw = arg0[2] & 0xFF;
                receivedData.altitude = arg0[3] & 0xFF;
                receivedData.whichThread = 1;
            } catch (Exception e) { }
        }
};
```

setSerialPort

```
public void setSerialPort(UsbManager usbManager, UsbDevice device){
    this.device = device;
    this.usbManager = usbManager;
```

```
    connection = this.usbManager.openDevice(this.device);
    serialPort = UsbSerialDevice.createUsbSerialDevice(this.device, connection);
    if (serialPort != null) {
        if (serialPort.open()) { //Set Serial Connection Parameters.
            serialPort.setBaudRate(115200);
            serialPort.setDataBits(UsbSerialInterface.DATA_BITS_8);
            serialPort.setStopBits(UsbSerialInterface.STOP_BITS_1);
            serialPort.setParity(UsbSerialInterface.PARITY_NONE);
            serialPort.setFlowControl(UsbSerialInterface.FLOW_CONTROL_OFF);
            serialPort.read(usbReadCallback);
            Log.d("Serial", "Serial Connection Opened");
            threadComputeFeedback = new Thread(runFeedbackThread);
            threadComputeFeedback.start();
        } else {
            Log.d("SERIAL", "PORT NOT OPEN");
        }
    } else {
        Log.d("SERIAL", "PORT IS NULL");
    }
}
```

setPowers

```
public void setPowers(MotorsPowers powers) {
    // Prepare data to send.
    txBuffer[0] = (byte) powers.sw;
    txBuffer[1] = (byte) powers.se;
    txBuffer[2] = (byte) powers.ne;
    txBuffer[3] = (byte) powers.nw;
    serialPort.write(txBuffer);
    if(!myTurn)
        myTurn = true;
}
```

A.1.2 Autopilot

setK

```
public void setK(float kp, float ki, float kd){
    yawRegulator.setCoefficients(0.8f, 0.1f, 0.3f);
    pitchRegulator.setCoefficients(0.8f, 0.1f, 0.3f);
    rollRegulator.setCoefficients(0.8f, 0.1f, 0.3f);
    altitudeRegulator.setCoefficients(0.8f, 0.1f, 0.3f);
}
```

compute

```
//whichThread 0 if feedbackThread 1 RCThread
public synchronized void compute(AdkCommunicator.ReceivedData receivedData){
    sensorsData = mySensors.getSensorsData();
    double tempPowerNW = 0, tempPowerNE = 0, tempPowerSE = 0, tempPowerSW = 0;
    if((inTheRange(receivedData.roll, 5, 120)) &&
        (inTheRange(receivedData.pitch, 5, 120))  &&
        (inTheRange(receivedData.yaw, 10, 235)) &&
        (inTheRange(receivedData.altitude, 5, 10)) &&
        receivedData.whichThread == 1){
        //Disarm
        Log.d("Disarm","Motors Disarmed");
        mySensors.sendArmDisarm(false);
    }
    else if((inTheRange(receivedData.roll, 10, 120)) &&
            (inTheRange(receivedData.pitch, 10, 120))  &&
            (inTheRange(receivedData.yaw, 10, 10)) &&
            (inTheRange(receivedData.altitude, 10, 10)) &&
            receivedData.whichThread == 1){
        //Arm
        Log.d("Arm","Motors Armed");
        mySensors.sendArmDisarm(true);
    }
    BigDecimal currentRoll = new BigDecimal(sensorsData.roll)
        .setScale(0, BigDecimal.ROUND_HALF_UP);
    BigDecimal targetAngleRoll = new BigDecimal(getAngle(receivedData.roll))
        .setScale(0, BigDecimal.ROUND_HALF_UP);
    BigDecimal currentPitch = new BigDecimal(sensorsData.pitch)
```

```
    .setScale(0, BigDecimal.ROUND_HALF_UP);
BigDecimal targetAnglePitch = new BigDecimal(getAngle(receivedData.pitch))
    .setScale(0, BigDecimal.ROUND_HALF_UP);
BigDecimal currentYaw = new BigDecimal(sensorsData.yaw)
    .setScale(0, BigDecimal.ROUND_HALF_UP);
BigDecimal targetAngleYaw = new BigDecimal(getAngle(receivedData.yaw))
    .setScale(0, BigDecimal.ROUND_HALF_UP);
float dt = ((float)
    (sensorsData.time - sensorsData.previousTime)) / 1000000000.0f; // [s].
if(receivedData.whichThread == 1) {
    Log.d(TAG, "RC");
    rollForce = rollRegulator
        .getInput(-targetAngleRoll.floatValue(), currentRoll.floatValue(), dt);
    pitchForce = pitchRegulator
        .getInput(-targetAnglePitch.floatValue(), currentPitch.floatValue(), dt);
    yawForce = yawRegulator
        .getInput(-targetAngleYaw.floatValue() + currentYaw.floatValue(),
            currentYaw.floatValue(), dt);
}
else if(receivedData.whichThread == 0){
    Log.d(TAG, "Feedback");
    rollForce = rollRegulator
        .getInput(targetAngleRoll.floatValue(), currentRoll.floatValue(), dt);
    pitchForce = pitchRegulator
        .getInput(targetAnglePitch.floatValue(), currentPitch.floatValue(), dt);
    yawForce = 0;
}
BigDecimal forceRoll = new BigDecimal(rollForce)
    .setScale(1, BigDecimal.ROUND_HALF_UP);
BigDecimal forcePitch = new BigDecimal(pitchForce)
    .setScale(1, BigDecimal.ROUND_HALF_UP);
BigDecimal forceYaw = new BigDecimal(yawForce)
    .setScale(1, BigDecimal.ROUND_HALF_UP);
altitudeForce = receivedData.altitude;
tempPowerNW = altitudeForce; // Vertical "force".
tempPowerNE = altitudeForce; //
tempPowerSE = altitudeForce; //
tempPowerSW = altitudeForce;
tempPowerNW += forceRoll.floatValue(); // Roll "force".%.2f
tempPowerNE -= forceRoll.floatValue(); //
tempPowerSE -= forceRoll.floatValue(); //
tempPowerSW += forceRoll.floatValue(); //
tempPowerNW += forcePitch.floatValue(); // Pitch "force".
tempPowerNE += forcePitch.floatValue(); //
tempPowerSE -= forcePitch.floatValue(); //
tempPowerSW -= forcePitch.floatValue(); //
tempPowerNW += forceYaw.floatValue(); // Yaw "force".
tempPowerNE -= forceYaw.floatValue(); //
tempPowerSE += forceYaw.floatValue(); //
tempPowerSW -= forceYaw.floatValue(); //
motorsPowers.nw = motorSaturation(tempPowerNW);
motorsPowers.ne = motorSaturation(tempPowerNE);
motorsPowers.se = motorSaturation(tempPowerSE);
motorsPowers.sw = motorSaturation(tempPowerSW);
String data = String.valueOf(SystemClock.elapsedRealtime())+","+
        String.valueOf(motorsPowers.nw)+","+
        String.valueOf(motorsPowers.ne)+","+
        String.valueOf(motorsPowers.se)+","+
        String.valueOf(motorsPowers.sw)+","+
        String.valueOf(altitudeForce)+","+
        String.valueOf(currentRoll)+","+
        String.valueOf(targetAngleRoll)+","+
        String.valueOf(forceRoll.floatValue())+","+
        String.valueOf(currentPitch)+","+
        String.valueOf(targetAnglePitch)+","+
        String.valueOf(forcePitch.floatValue())+","+
        String.valueOf(currentYaw)+","+
        String.valueOf(targetAngleYaw)+","+
        String.valueOf(forceYaw.floatValue())+",\n";
try {
    outStream.write(data.getBytes());
} catch (IOException e) {
    e.printStackTrace();
```

```
    }
    adkCommunicator.setPowers(motorsPowers);
}
```

A.1.3 MySensors

onSensorChanged

```
@Override
public void onSensorChanged(SensorEvent event) {
    if(event.sensor.getType() == Sensor.TYPE_ROTATION_VECTOR){
        ....
        // Convert the to "yaw, pitch, roll".
        SensorManager.getRotationMatrixFromVector(rotationMatrix, rotationVec);
        SensorManager.getOrientation(rotationMatrix, yawPitchRollVec);
        sensorsData.yawTemp =
            getMainAngle(-(yawPitchRollVec[0]) * RAD_TO_DEG);
        sensorsData.pitchTemp =
            getMainAngle(-(yawPitchRollVec[1]-pitchZero) * RAD_TO_DEG);
        sensorsData.rollTemp =
            getMainAngle((yawPitchRollVec[2]-rollZero) * RAD_TO_DEG);
        if (dataRoll != null &&
            dataPitch != null &&
            dataYaw != null) {
            ILineDataSet setRoll = dataRoll.getDataSetByIndex(0);
            ILineDataSet setPitch = dataPitch.getDataSetByIndex(0);
            ILineDataSet setYaw = dataYaw.getDataSetByIndex(0);
            ...
            //Roll
            dataRoll.addEntry(
                new Entry(setRoll.getEntryCount(), sensorsData.rollTemp), 0);
            dataRoll.notifyDataChanged();
            // let the chart know it's data has changed
            rollChart.notifyDataSetChanged();
            // limit the number of visible entries
            rollChart.setVisibleXRangeMaximum(20);
            // move to the latest entry
            rollChart.moveViewToX(dataRoll.getEntryCount());
            //Pitch
            dataPitch.addEntry(
                new Entry(setPitch.getEntryCount(), sensorsData.pitchTemp), 0);
            ....
            //Yaw
            dataYaw.addEntry(
                new Entry(setYaw.getEntryCount(), sensorsData.yawTemp), 0);
            ...
        }
    }
    else if(event.sensor.getType() == Sensor.TYPE_PRESSURE){
        ...
        float pressure = event.values[0];
        float rawAltitudeUnsmoothed = SensorManager
            .getAltitude(SensorManager.PRESSURE_STANDARD_ATMOSPHERE, pressure);
        absoluteElevation = (absoluteElevation*ALTITUDE_SMOOTHING) +
            (rawAltitudeUnsmoothed*(1.0f-ALTITUDE_SMOOTHING));
        sensorsData.altitudeTemp = absoluteElevation - elevationZero;
        dataAltitude.addEntry(
            new Entry(setAltitude.getEntryCount(), sensorsData.altitudeTemp), 0);
        ...
     }
    if(sensorsData.count < NUMBER_OF_SAMPLE) {
        sensorsData.accumulationYaw += sensorsData.yawTemp;
        sensorsData.accumulationRoll += sensorsData.rollTemp;
        sensorsData.accumulationPitch += sensorsData.pitchTemp;
        sensorsData.accumulationAltitude += sensorsData.altitudeTemp;
        sensorsData.count++;
    }
    else if(sensorsData.count == NUMBER_OF_SAMPLE){
        ...
        sensorsData.previousTime = sensorsData.time;
        sensorsData.time = event.timestamp;
        sensorsData.roll = sensorsData.accumulationRoll/NUMBER_OF_SAMPLE;
        sensorsData.pitch = sensorsData.accumulationPitch/NUMBER_OF_SAMPLE;
        sensorsData.yaw = sensorsData.accumulationYaw/NUMBER_OF_SAMPLE;
        sensorsData.altitude = sensorsData.accumulationAltitude/NUMBER_OF_SAMPLE;
        ...
        this.connectionMavlink.sendToAllSensorsData(sensorsData);
        ...
```

```
    }
}
```

A.1.4 PidAngleRegulation

getInput

```
public float getInput(float targetAngle, float currentAngle, float dt)
{
    // The complete turn can be done, so we have to be careful around the
    // +180, -180 limit.
    float rawDifference = targetAngle - currentAngle;
    float difference = getMainAngle(rawDifference);
    //Lock angle + or - 30 degrees. What do you do with yaw?
    if(difference > MAX_ROLL_PITCH)
        difference = MAX_ROLL_PITCH;
    else if (difference < -MAX_ROLL_PITCH)
        difference = -MAX_ROLL_PITCH;
    boolean differenceJump = (difference != rawDifference);
    // Now, the PID computation can be done.
    float input = 0.0f;
    // Proportional part.
    input += difference * kp;//la meta' della differenza
    // Integral part.
    integrator += difference * dt * ki;
    input += integrator;
    // Derivative part, with filtering.
    if(!differenceJump)
    {
        differencesMean = differencesMean * smoothingStrength
                + difference * (1-smoothingStrength);
        float derivative = (differencesMean - previousDifference) / dt;
        previousDifference = differencesMean;
        input += derivative * kd;
    }
    else
    {
        // Erase the history, because we are not reaching the target from
        // the "same side".
        differencesMean = 0.0f;
    }
    return input;
}
```

A.1.5 PidRegulation

getInput

```
public float getInput(float targetAngle, float currentAngle, float dt)
{
    float difference = targetAngle - currentAngle;
    // Now, the PID computation can be done.
    float input = aPriori;
    // Proportional part.
    input += difference * kp;
    // Integral part.
    integrator += difference * ki * dt;
    input += integrator;
    // Derivative part, with filtering.
    differencesMean = differencesMean * smoothingStrength
            + difference * (1-smoothingStrength);
    float derivative = (differencesMean - previousDifference) / dt;
    previousDifference = differencesMean;
    input += derivative * kd;
    return input;
}
```

A.2 androidGroundControl Package

A.2.1 Connection

run

```
@Override
public void run() {
try (ServerSocket serverSocket = new ServerSocket(5760)) {
    ...
    sendDataToGCS = new SendDataToGCS(systemId, componentId);
    while(true){
        Socket socket = serverSocket.accept();
        MavlinkConnection connection = MavlinkConnection.create(
                socket.getInputStream(),
                socket.getOutputStream());
        connections.add(connection);
        sendDataToGCS.sendBatteryStatus(connection, batteryLevel);
        sendDataToGCS.sendGpsData(connection, actualLocation);
        new Thread(new Runnable() {
            @Override
            public void run() {
                Log.d("HeartBeat", "send");
                sendDataToGCS.sendHeartbeat(connection);
            }
        }).start();
        new Thread(new Runnable() {
            @Override
            public void run() {
                MavlinkMessage message;
                Intent serviceTakePhoto = new Intent(mainContext, TakePhoto.class);
                int count = 0;
                try{
                    while((message = connection.next())!= null){
                        if(message.getPayload() instanceof CommandLong){
                            CommandLong c = (CommandLong) message.getPayload();
                            if(c.command().entry() ==
                                    MavCmd.MAV_CMD_DO_DIGICAM_CONTROL) {
                                if(c.param5() == 1.0 || c.param7() == 1.0){
                                    count++;
                                    CommandLong commandLong = CommandLong.builder()
                                        .command(MavCmd.MAV_CMD_IMAGE_START_CAPTURE)
                                        .targetComponent(100)
                                        .targetSystem(1)
                                        .param3(1)
                                        .build();
                                    CommandAck commandAck = CommandAck.builder()
                                        .command(MavCmd.MAV_CMD_DO_DIGICAM_CONTROL)
                                        .targetSystem(255)
                                        .targetComponent(244)
                                        .build();
                                    CameraTrigger cameraTrigger =
                                    CameraTrigger.builder()
                                    .seq(count)
                                        .timeUsec(new BigInteger(64, new Random(55)))
                                        .build();
                                    new Thread(new Runnable() {
                                        @Override
                                        public void run() {
                                            ((AppCompatActivity)mainContext)
                                                .startService(serviceTakePhoto);
                                        }
                                    }).start();
                                    connection
                                        .send2(systemId, componentId, commandLong);
                                    connection
                                        .send2(systemId, componentId, commandAck);
                                    connection
                                        .send2(systemId, componentId, cameraTrigger);
                                }
                            }
                            else if(c.command().entry().equals(
                                MavCmd.MAV_CMD_REQUEST_AUTOPILOT_CAPABILITIES)){
                                AutopilotVersion autopilotVersion =
                                    AutopilotVersion.builder()
                                        .capabilities(MavProtocolCapability
                                            .MAV_PROTOCOL_
```

```
                                                    CAPABILITY_FLIGHT_INFORMATION)
                                            .capabilities(MavProtocolCapability
                                                .MAV_PROTOCOL_
                                                    CAPABILITY_MAVLINK2)
                                            .capabilities(MavProtocolCapability
                                                .MAV_PROTOCOL_
                                                CAPABILITY_COMMAND_INT)
                                            .osSwVersion(222)
                                            .flightSwVersion(111)
                                            .boardVersion(333)
                                            .productId(1)
                                            .build();
                                    ProtocolVersion protocolVersion =
                                        ProtocolVersion.builder()
                                            .maxVersion(300)
                                            .minVersion(100)
                                            .version(300)
                                            .build();
                                    connection.send2(systemId, componentId,
                                        autopilotVersion);
                                    connection.send2(systemId, componentId,
                                        protocolVersion);
                                }
                            }
                            ...
                        }
                    }
                }
            }).start();
    }
...
}
}
```

sendToAllGpsData

```
public void sendToAllGpsData(Location location) {
        for(MavlinkConnection connection : connections)
            sendDataToGCS.sendGpsData(connection, location);
}
```

sendToAllBatteryStatus

```
public void sendToAllBatteryStatus(int level) {
    for(MavlinkConnection connection : connections)
        sendDataToGCS.sendBatteryStatus(connection, level);
}
```

sendToAllSensorsData

```
public void sendToAllSensorsData(MySensors.SensorsData sensorsData) {
    for(MavlinkConnection connection : connections)
        sendDataToGCS.sendSensorsData(connection, sensorsData);
}
```

A.2.2 SendDataToGCS

sendSensorsData

```
public void sendSensorsData(MavlinkConnection connection,
         MySensors.SensorsData sensorsData){
        Attitude attitude = Attitude.builder()
            .yaw(-sensorsData.yaw/180*PI)
            .pitch(-sensorsData.pitch/180*PI)
            .roll(-sensorsData.roll/180*PI)
            .timeBootMs(SystemClock.elapsedRealtime())
            .build();
    Altitude altitude =
    Altitude.builder().altitudeRelative(sensorsData.altitude)
        .build();
    new Thread(new Runnable() {
        @Override
        public void run() {
            try {
                if(connection!=null ) {
                    connection.send2(systemId,
                        componentId, attitude);
                    connection.send2(systemId,
```

```
                        componentId, altitude);
                }
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }).start();
}
```

sendBatteryStatus

```
public void sendBatteryStatus(
        MavlinkConnection connection, int level){
    SysStatus sysStatus = SysStatus.builder()
        .batteryRemaining(level).build();
    new Thread(new Runnable() {
        @Override
        public void run() {
            try {
                if(connection!=null) {
                    connection.send2(systemId,
                        componentId, sysStatus);
                }
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }).start();
}
```

sendGpsData

```
public void sendGpsData(MavlinkConnection connection,
    Location location){
    if(location != null){
        GpsRawInt gps = GpsRawInt.builder()
                .lat((int)Math
                    .round(location.getLatitude() * 10000000))
                .lon((int)Math
                    .round(location.getLongitude() * 10000000))
                .alt((int)Math
                    .round(location.getAltitude() * 1000))
                .timeUsec(BigInteger
                    .valueOf(location.getTime()))
                .fixType(GpsFixType.values())
                .build();
        new Thread(new Runnable() {
            @Override
            public void run() {
                try {
                    if(connection!=null)
                        connection.send2(systemId,
                            componentId, gps);
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }).start();
    }
}
```

sendHeartbeat

```
public void sendHeartbeat(MavlinkConnection connection){
    Heartbeat heartbeat;
    int systemId = 2;
    int componentId = 1;
    boolean connectionActive = true;
    while (connectionActive) {
        try {
            sleep(1000);
            heartbeat = Heartbeat.builder()
                    .type(MavType
                        .MAV_TYPE_QUADROTOR)
                    .autopilot(MavAutopilot
                        .MAV_AUTOPILOT_GENERIC)
                    .baseMode(mavMode)
```

```
                    .systemStatus(MavState
                        .MAV_STATE_ACTIVE)
                    .mavlinkVersion(2)
                    .build();
            connection.send2(systemId,
                componentId, heartbeat);
        } catch (IOException e) {
            e.printStackTrace();
            connectionActive = false;
        } catch (InterruptedException e) {
            e.printStackTrace();
            connectionActive = false;
        }
    }
}
```

A.2.3 TakePhoto

```
@RequiresApi(api = Build.VERSION_CODES.N)
@Override
public int onStartCommand(Intent intent, int flags, int startId) {
    PixelCopy.OnPixelCopyFinishedListener callback =
        new PixelCopy.OnPixelCopyFinishedListener() {
        @Override
        public void onPixelCopyFinished(int copyResult) {
        }
    };
    SurfaceView surfaceView = SessionBuilder.getInstance().getSurfaceView();
    Bitmap bitmap = Bitmap.createBitmap(surfaceView.getWidth(),
            surfaceView.getHeight(), Bitmap.Config.ARGB_8888);
    PixelCopy.request(surfaceView, bitmap, callback, handler);
    File sd = new File(getExternalFilesDir(
        Environment.DIRECTORY_PICTURES), "PictureOnAir");
    if(!sd.exists()) {
        sd.mkdirs();
        Log.i("FO", "folder" + getExternalFilesDir(
            Environment.DIRECTORY_PICTURES));
    }
    Calendar cal = Calendar.getInstance();
    SimpleDateFormat sdf = new SimpleDateFormat("yyyyMMdd_HHmmss");
    String tar = (sdf.format(cal.getTime()));
    try {
        FileOutputStream outStream = new FileOutputStream(sd+"/"+tar+".jpg");
        bitmap.compress(Bitmap.CompressFormat.PNG, 100, outStream);
        Log.d("Photo", "--->Photo Saved");
    } catch (FileNotFoundException e) {
        e.printStackTrace();
    }
    return super.onStartCommand(intent, flags, startId);
}
```

Appendix B

ArduinoCode

B.0.1 Setup

```
void setup() {
  // put your setup code here, to run once:
  Serial.begin(115200);
  pinMode(RX_ROLL, INPUT);
  pinMode(RX_YAW, INPUT);
  pinMode(RX_PITCH, INPUT);
  pinMode(RX_ALTITUDE, INPUT);
  pinMode(NW_PWM_PIN, OUTPUT);
  pinMode(NE_PWM_PIN, OUTPUT);
  pinMode(SW_PWM_PIN, OUTPUT);
  pinMode(SE_PWM_PIN, OUTPUT);
  Serial.setTimeout(1);
  lengthOfRx = 0;
}
```

B.0.2 Loop

```
void loop() {
  byte rx[256];
  usRxRoll = pulseIn(RX_ROLL, HIGH);
  usRxPitch = pulseIn(RX_PITCH, HIGH);
  usRxYaw = pulseIn(RX_YAW, HIGH);
  usRxAltitude = pulseIn(RX_ALTITUDE, HIGH);
  pwmValueRoll = map(usRxRoll, PULSE_MIN, PULSE_MAX, 0, 255);
  pwmValuePitch = map(usRxPitch, PULSE_MIN, PULSE_MAX, 0, 255);
  pwmValueYaw = map(usRxYaw, PULSE_MIN, PULSE_MAX, 0, 255);
  pwmValueAltitude = map(usRxAltitude, PULSE_MIN, PULSE_MAX, 0, 255);
  if((inTheRange(pwmValueRoll, 5, 120)) && (inTheRange(pwmValuePitch, 5, 120))
&& (inTheRange(pwmValueYaw, 10, 235)) && (inTheRange(pwmValueAltitude, 5, 10))){
      nwMotor.detach();
      neMotor.detach();
      swMotor.detach();
      seMotor.detach();
      nwPower = 0;
      swPower = 0;
      nePower = 0;
      sePower = 0;
    }
  else if((inTheRange(pwmValueRoll, 10, 120)) && (inTheRange(pwmValuePitch, 10, 120))
&& (inTheRange(pwmValueYaw, 10, 10)) && (inTheRange(pwmValueAltitude, 10, 10))){
    nwMotor.attach(NW_PWM_PIN);
    neMotor.attach(NE_PWM_PIN);
    swMotor.attach(SW_PWM_PIN);
    seMotor.attach(SE_PWM_PIN);
    nwPower = PULSE_MIN;
    swPower = PULSE_MIN;
    nePower = PULSE_MIN;
    sePower = PULSE_MIN;
  }
  if(!(inTheRange(pwmValueRoll, 5, 120)) || !(inTheRange(pwmValuePitch, 5, 120))
    || !(inTheRange(pwmValueYaw, 5, 120)) || !(inTheRange(pwmValueAltitude, 5, pwmPrecAlti
    pwmPrecAltitude = pwmValueAltitude;
    tx[0] = pwmValueRoll;
    tx[1] = pwmValuePitch;
```

```
    tx[2] = pwmValueYaw;
    tx[3] = pwmValueAltitude;
    Serial.write(tx, sizeof(tx));
  }
  if(Serial.available()){
    lengthOfRx = Serial.readBytes(rx, sizeof(rx));
    sw = (byte) rx[(lengthOfRx-4)];
    se = (byte) rx[(lengthOfRx-3)];
    ne = (byte) rx[(lengthOfRx-2)];
    nw = (byte) rx[(lengthOfRx-1)];
    nwPower = map(nw, 0, 255, PULSE_MIN, PULSE_MAX);
    nePower = map(ne, 0, 255, PULSE_MIN, PULSE_MAX);
    swPower = map(sw, 0, 255, PULSE_MIN, PULSE_MAX);
    sePower = map(se, 0, 255, PULSE_MIN, PULSE_MAX);
    nwMotor.writeMicroseconds(nwPower);
    neMotor.writeMicroseconds(nePower);
    swMotor.writeMicroseconds(swPower);
    seMotor.writeMicroseconds(sePower);
  }
}
```